IcedTea: Efficient and Responsive Time-Travel Debugging in Dataflow Systems (Full Version)

Shengquan Ni
UC Irvine
Irvine, CA, USA
shengqun@uci.edu

Yicong Huang
UC Irvine
Irvine, CA, USA
yicongh1@ics.uci.edu

Zuozhi Wang
UC Irvine
Irvine, CA, USA
zuozhiw@ics.uci.edu

Chen Li
UC Irvine
Irvine, CA, USA
chenli@ics.uci.edu

ABSTRACT

Dataflow systems have an increasing need to support a wide range of tasks in data-centric applications using latest techniques such as machine learning. These tasks often involve custom functions with complex internal states. Consequently, users need enhanced debugging support to understand runtime behaviors and investigate internal states of dataflows. Traditional forward debuggers allow users to follow the chronological order of operations in an execution. Therefore, a user cannot easily identify a past runtime behavior *after* an unexpected result is produced. In this paper, we present a novel time-travel debugging paradigm called IcedTea, which supports reverse debugging. In particular, in a dataflow's execution, which is inherently distributed across multiple operators, the user can periodically interact with the job and retrieve the global states of the operators. After the execution, the system allows the user to roll back the dataflow state to any past interactions. The user can use step instructions to repeat the past execution to understand how data was processed in the original execution. We give a full specification of this powerful paradigm, study how to reduce its runtime overhead and develop techniques to support debugging instructions responsively. Our experiments on real-world datasets and workflows show that IcedTea can support responsive time-travel debugging with low time and space overhead.

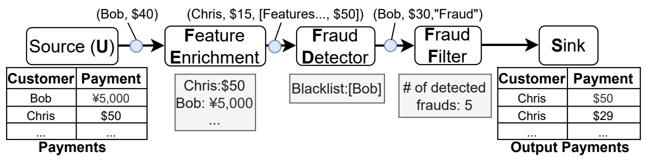
PVLDB Reference Format:
Shengquan Ni, Yicong Huang, Zuozhi Wang, and Chen Li. IcedTea: Efficient and Responsive Time-Travel Debugging in Dataflow Systems (Full Version). PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

Dataflow systems are widely used in modern data-centric applications. Many of these systems are designed for handling analytical workloads [2, 11, 30]. Recently, there is an emerging trend towards handling more diverse workloads, using the latest advanced techniques such as machine learning (ML) algorithms [6, 37] and event-driven cloud applications [4]. These advanced use cases often involve user-defined functions (UDFs) [29, 31, 36] with complex implementation logic and internal states. The rise in the complexity of dataflow jobs imposes a significant need for good debugging support. Users increasingly want to understand a dataflow's runtime behaviors and identify potential issues in both data and the dataflow logic itself.

As an example, consider a simple dataflow in Figure 1 designed to detect fraudulent credit card transactions. The Source operator (U for short) emits the tuples in a Payments table one by one to the Feature Enrichment (FE for short) operator. The latter generates features for a payment and sends the payment with the features to the next operator. One of the features is the highest payment of the customer seen by FE so far. The next Fraud Detector operator (FD) uses a machine learning model based on the features to identify fraudulent transactions and marks the output transactions with either a "Fraud" or an "Approved" label. Internally it maintains a blacklist of customers. If a payment transaction is identified as fraudulent, its customer is added to the blacklist, and all subsequent transactions of the same customer will be flagged as fraud. The Fraud Filter operator (FF) drops the fraudulent transactions and keeps track of the number of fraudulent transactions. The Sink (S) operator collects all non-fraudulent transactions as the final output.



Customer	Payment
Bob	¥5,000
Chris	$50
...	...

Customer	Payment
Chris	$50
Chris	$29
...	...

Figure 1: A fraud-detection workflow uses features including the highest payment of a customer to identify fraudulent transactions and drop them. A bug causes the payments of Bob to be filtered unexpectedly.

Suppose after an execution of the dataflow, a user notices that some transactions that are expected to be approved do not appear in the outputs. The user believes there should be a bug in the dataflow, possibly in one of the operators, but cannot pinpoint its location.

Existing debugging solutions and limitations. There are two common methods to help the user find the bug. One method is to support runtime debugging [17, 19, 23], which allows the user to set breakpoints and pause or resume the execution, akin to using a traditional language debugger such as gdb for C/C++ and jdb for Java. The user can inspect the state of a suspicious operator *after* an issue has happened, thus it is known as "post-mortem" debugging. A main limitation of this method is that the runtime behaviors that caused the unexpected result already happened, but the user cannot

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

go back to the past to identify the behavior. To find the root case, the user may have to rerun the entire dataflow from the beginning, hoping to repeat and locate the runtime behavior. This method not only needs to repeat the previous execution (which can be computationally expensive), but also lacks the guarantee to reproduce the problematic runtime behavior in the original execution. Another method is to use data lineage and provenance to trace the transformations of data tuples (e.g., [10, 20, 34]). It helps the user trace backward to identify where data went wrong in the execution and understand the origin of the problematic tuple. A main limitation of this approach is that it does not capture how the state of each operator evolved during execution, while the state can be crucial for the user to understand the runtime behavior and identify a bug. In the running example, the user may need to examine the internal states of the FE and FD operators to understand why the problematic tuple is mistakenly marked as fraud. In this case, analyzing the lineage of the problematic tuple alone is insufficient to find the bug, which could be related to a logic error or state error within the operators. Details about the bug will be explained in Section 3.

Proposed solution. Notice that a similar problem exists in programming languages, and reverse debuggers [22, 27] allow users to repeat a program's past execution to debug. These tools [28, 33] have shown to be very powerful in helping programmers find bugs in code. In this paper, we study how to support a similar experience in the execution of dataflows. We develop a novel debugging paradigm called "IcedTea"[1], which allows users to interact with an execution of dataflow to inspect its state and later faithfully repeat the execution to enable *time travel* to past interactions. Figure 2 gives an overview of the system. After the original execution, the user can jump to any of the past interactions, particularly those before unexpected results were generated (e.g., time 10:05). In addition, the system allows the user to control the replay process and take steps to inspect how the operator states changed.

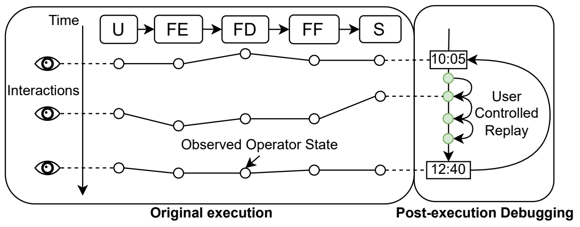


Figure 2: User experience of time-travel debugging with IcedTea. In an original execution, users interact with the workflow to inspect operator states. After the execution, users return to a past interaction and replay the execution.

Requirements and challenges. First, the execution of a workflow with multiple operators is inherently distributed. As debugging aims to identify bugs related to *data processing*, we need to capture meaningful global states of the operators in this distributed environment to help the user understand the lifecycle of data tuples. Second, debugging primitives need to consider the workflow's DAG structure and how data records are processed through the operators in the DAG. Third, a log-based method to support the repeatability of the original execution needs to have low overhead in terms of both time and space, especially when the data volume is large. Fourth, since debugging is a user-facing experience, each time-travel request should be served responsively, ideally within seconds or even milliseconds.

[1]It stands for "**I**nterac**t**ive **e**xecutions of **d**ataflows in **T**im**e**-tr**a**vel debugging."

Paper organization and contributions. The following is the organization of the paper with our contributions. In Section 2 we present a dataflow architecture and computation model of each operator in IcedTea. In Section 3, we explain how a user interacts with a dataflow execution and develop a new concept called "tuple-consistent snapshot" for the operators, which is useful for the user to understand how data tuples are processed during the execution. In Section 4, we study how to allow the user to time travel after the execution. We develop debugging primitives including jumps and step operations, and how to do logging and use the log records to support these primitives. In Section 5, we extend the results to general dataflow DAGs and discuss how to support deterministic replay in the presence of non-determinism within operators. In Section 6 we study how to support the debugging primitives responsively, with considerations of time and space overhead. In Section 7 we present experimental results to evaluate IcedTea on real-world datasets and workflows.

1.1 Related Work

Debugging in dataflow systems. For instance, BigDebug [17] introduces simulated breakpoints for Spark-based [1] applications. Amber [23] allows developers to pause and resume a dataflow job and set conditional breakpoints. Port [25] supports an isolated environment for debugging sessions by transferring them to an external process. Udon [19] focuses on runtime debugging on user-defined functions, and it allows users to set breakpoints on code lines and control the execution line-by-line. IcedTea is different from these debugging techniques, as it can support "backward debugging" by allowing the user to go back to the past in the original execution. Ambrosia [15] provides record-and-replay capabilities to support time-travel debugging, and during the replay phase, it attaches a debugger to a *single node* to inspect its internal state. In contrast, IcedTea supports time-travel debugging on *multiple nodes*, allowing users to trace a tuple's processing across multiple operators. It guarantees a tuple-consistent snapshot across multiple operators, which can better help users debug dataflow jobs.

Data linage and provenance. Data lineage helps users identify potential errors by showing how data is processed through various stages in a dataflow. Provenance provides historical metadata about data, such as its origins, changes, and context. For instance, TagSniff [10] presents a tag-based system to label and trace data between operators. OptDebug [18] develops a method for pinpointing fault-inducing operations by streamlining input records and tracking operation lineage. Titan [20] integrates data provenance into Apache Spark [1] and allows users to trace the lineage of data through Spark transformations. Provenance-based approaches do not allow users to recover the state of an operator back in a past execution of a dataflow. In contrast, IcedTea allows users to revert the states of operators to an earlier time point of a past execution and start inspection from that point. In Section 7, we show that many operators in real-world workflows are stateful.

Reverse debuggers in programming languages. Such debuggers have proven to be powerful in various contexts. For instance, GDB [14] supports record-and-replay debugging. There are commercial solutions such as UndoDB [33] (for C/C++ programs) and Replay [28] (for Web applications). Reverse debuggers for general programs need to handle various non-deterministic factors, such as I/O on files or networks and concurrent multi-core executions. The recording process in these tools could have a significant overhead [9]. Compared to these solutions, IcedTea is designed for dataflow systems, where an execution is distributed. The solution is different as it utilizes the DAG structure and data-processing properties, which are not a focus of traditional reverse debuggers.

Existing snapshot algorithms. Many existing methods, such as the Chandy-Lamport algorithm [7] and Flink's asynchronous snapshotting algorithm [3], focus on capturing a globally consistent state in a distributed system, to support fault tolerance. These approaches require capturing the states of all the operators in a dataflow. IcedTea uses a different snapshot semantic in order to trace the lifecycle of the processing of an input tuple. In particular, for each input tuple of an operator, IcedTea captures the states of the operator and all its downstream operators, and ignores the states of other irrelevant operators.

2 OVERVIEW OF ICEDTEA

2.1 Dataflow Systems

A dataflow (a.k.a. *workflow*) is a directed acyclic graph (DAG) of operators, denoted as $W = (V, E)$, where V is a set of operators, and E is a set of uni-directional *edges* connecting two operators. An operator has internal variables, e.g., a total count in a COUNT operator. It processes input messages, updates its variables, and generates output messages. We consider a *pipelined-execution* model in which multiple operators run in parallel. Without waiting to receive all input data, an operator can output its results so that its downstream operators can process the data simultaneously. During the execution of the workflow, a module called *coordinator* receives interaction requests from users and communicates with the operators. The coordinator has a bidirectional *channel* with each operator, and the channels and edges between operators guarantee first-in, first-out (FIFO) and exactly-once message delivery. An edge or channel assigns an incrementally increasing sequence number to each of its messages. There are two types of messages, namely *tuple messages* (or "tuples" for short) and *control messages*. A tuple message includes data to be processed in the workflow. A control message includes an instruction to retrieve the internal state of an operator (defined shortly). Tuples are transferred through edges between operators, while control messages are sent via the channel between the coordinator and an operator or on edges between operators. On the edges, control messages are transferred together with tuples following their FIFO order.

Figure 3 shows how the two types of messages are transferred during the execution of the example workflow. The coordinator sends control messages to all the operators and waits for control messages from them. For instance, the coordinator sends a control message c to the FE operator to retrieve a mapping of customers to their highest payments.

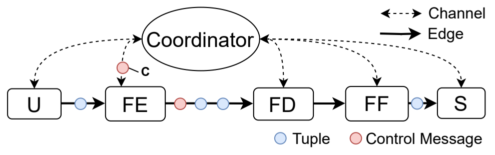


Figure 3: During the execution of the workflow, the coordinator sends control messages to operators and receives control messages from operators through channels. Operators send control messages and tuples through edges.

2.2 Computation of an Operator

During a dataflow execution, the operators do the computation in discretized steps that can change their internal states. Between two discretized steps, the operator is considered to be *waiting*.

Definition 2.1 (State and computation step of an operator). A *state* of an operator in a workflow includes values of its internal variables. In each *computation step*, the operator consumes an input message (a control message or a tuple), updates its state, and generates output messages.

Figure 4 shows the state transitions of three computation steps of the Feature Enrichment operator in the running example. The operator processes a tuple or a control message in each computation step. The operator's state includes the highest payment for each customer. Initially, the state has Bob's highest payment of \$5, and Chris' highest payment of \$40. In the first computation step, the operator processes a tuple $x = (Bob, \$20)$, and updates its state by changing Bob's value to \$20. It then emits a tuple x_1 with Bob's current highest payment, \$20, appended as the third field. When processing a tuple $y = (Chris, \$10)$, the operator does not alter its state, and it emits a tuple y_1 with Chris' current highest payment, \$40. After processing y, the operator processes a control message c from the coordinator and sends its state as another control message $c1$ back to the coordinator without modifying its state.

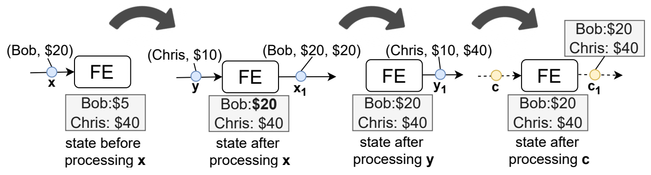


Figure 4: Three computation steps of the FE operator.

2.3 User Experience

As illustrated in Figure 2, during an execution of a workflow in IcedTea, periodically the user sends an interaction request to the system to retrieve the runtime states of the operators without pausing or stopping the execution. After the execution, the user can roll back the workflow to the states of any of the past interactions by sending a "jump" instruction. After that, the user can perform a controlled replay using "step" instructions to replicate the original execution and investigate how the states of the operators evolved throughout the process. This way, the user can understand past runtime behaviors and identify bugs.

3 ORIGINAL EXECUTIONS WITH INTERACTIONS

In this section, we present how users interact with a dataflow during its original execution to retrieve a global state of the operators. We use an example to show the limitation of the classic "happen-before consistency." We present a new concept called "tuple-based consistency" and develop an algorithm for retrieving a global state with this type of consistency.

3.1 Interaction and State Snapshots

Before executing a dataflow W, a user specifies an operator as an *interesting operator*, denoted as θ. Let $G(\theta)$ be the sub-DAG of W that includes θ and its downstream operators and edges. During execution, the system allows the user to retrieve the state of operators in $G(\theta)$. Additionally, post-execution debugging on these operators is supported to understand their runtime behavior. In the running example, if the user chooses FE as the interesting operator, then $G(\text{FE})$ includes the sub-DAG with operators FE, FD, FF, and S.

Definition 3.1 (Snapshot). During an execution of a dataflow W, a *snapshot* starting at an interesting operator θ is a set of states of the operators in the sub-DAG $G(\theta)$, denoted as $S(W, \theta)$.

Definition 3.2 (Runtime Interaction). Given an interesting operator θ of a dataflow W, a *runtime interaction* during an execution of W is a request to retrieve a snapshot starting at the operator θ.

An interaction can be triggered by the user or the system. For system-triggered interactions, the user sets conditions on input tuples of the interesting operator. If a tuple meets the conditions, the interesting operator requests an interaction. Dataflow execution is distributed, with multiple operators running in parallel. We ensure a retrieved snapshot for a user interaction is "consistent." A common consistency notion is based on "happen-before" [24]. This requires that if the state of a process reflects a message receipt, the state of the sender reflects the message sending [12]. Algorithms like Chandy-Lamport [7] retrieve happen-before consistent snapshots.

To illustrate happen-before consistency's limitation in debugging tuple-oriented execution, Figure 5 shows four interactions $I_1, \ldots, I_4$ during dataflow execution. Suppose the user sets a condition to monitor tuples entering FE with amounts over 1,000. Interaction I_3 is triggered by the input tuple z due to its non-dollar currency. Each interaction retrieves a snapshot satisfying happen-before consistency. At I_1, the snapshot shows all operators in their initial state. The tuple x, which contains Bob's payment of \$20, is the next tuple that FE will process. At I_2, the snapshot shows FE after processing x and producing x_1, adding the \$20 to its state. The next tuple y contains Chris's \$10 payment. At time t_z, a tuple with Bob's ¥5,000 payment is sent to FE. Interaction I_3 is triggered after FE processes z. The snapshot at I_3 shows FE after adding Chris's payment and updating Bob's largest payment to ¥5,000. Later, the user requests interaction I_4, which shows that Bob is blacklisted with his highest payment being ¥5,000. It also reveals 23 transactions marked as fraud on FF, all attributed to Bob. This is unexpected since ¥5,000 is a small amount. The user wonders if Bob was blacklisted when the ¥5,000 payment was processed, but this cannot be confirmed from the snapshot at I_3, where FD's blacklist shows nothing.

Since a happen-before consistent snapshot shows states while a tuple is still "going through" operators, the user may not see the complete effect of the tuple. In our example, although tuple z triggers an interaction, the retrieved snapshot does not reveal the bug caused by this tuple. Since the payment is not processed by FD, the user cannot determine if it will blacklist Bob. To address this limitation, we introduce a new kind of consistency.

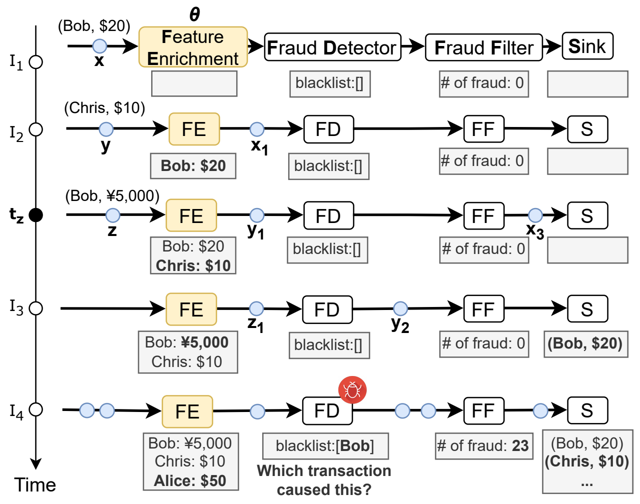


Figure 5: At I_4, the user notices that Bob is incorrectly blacklisted since his largest payment is ¥5,000, which is considered a small payment. This bug is caused by tuple z. Although the tuple arrived at time t_z, the snapshot at I_3 did not reveal the bug because the tuple has not been processed by FE.

3.2 Tuple-Based Consistency

To address this limitation, we introduce a new notion of consistency called "tuple-consistency" in the context of tuple processing. Intuitively, a tuple-consistent snapshot captures the states of the operators in $G(\theta)$ after all tuples into the interesting operator θ are completely processed by the sub-DAG. For simplicity, we assume the downstream operators of the interesting operator form a chain. We will relax this assumption in Section 5.

Definition 3.3 (Snapshot after a sequence of tuples). Given a sequence of tuples T to an interesting operator θ, the snapshot of $G(\theta)$ after T, denoted as $H(W, \theta, T)$, includes the operator states in $G(\theta)$. For θ, its state reflects that it processed all tuples in T and no more tuples are processed. For the states of other operators in $G(\theta)$, their states reflect that every tuple generated through the processing of T has been processed.

Figure 6 shows the comparison between the snapshot at I_3 and a snapshot I_3' after a sequence of tuples x, y and z. The latter shows that Bob has been blacklisted. Because tuple y and z are fully processed by all the operators, the user can gain insights that at the time when FE updates the largest transaction for Bob to ¥5,000, Bob is added to the blacklist.

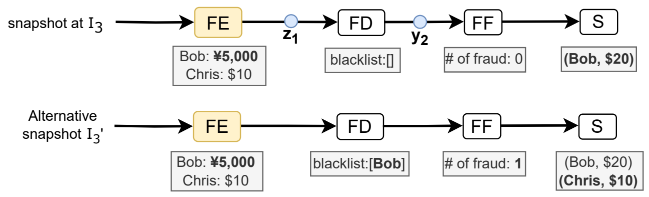


Figure 6: The comparison between the happen-before snapshot at I_3 and an alternative snapshot I'_3 after a set of tuples x, y and z. The latter is more informative. It shows that Bob is added to the blacklist because of his ¥5,000 payment.

Definition 3.4 (Tuple Consistency). Given an interesting operator θ in a dataflow W, a snapshot $S(W, \theta)$ is *tuple-consistent* if there exists a sequence of tuples T such that $S(W, \theta) = H(W, \theta, T)$.

The alternative snapshot I'_3 in Figure 6 is tuple-consistent with respect to a set of tuples x, y and z. Tuple-consistent snapshots can better help the user understand the states after a set of tuples is processed by the operators.

3.3 Retriving Tuple-Consistent Snapshots

We propose an algorithm for retrieving a tuple-consistent snapshot without interrupting the execution of the dataflow. Its main idea is to insert a barrier between the processing of two consecutive tuples of an interesting operator. In this way, the barrier separates the processing of two tuples. Operators report their states after receiving barriers from their upstream operators.

The algorithm works as follows. Given an interesting operator θ, the coordinator sends a control message to θ, which propagates the control message to its downstream operators. Algorithm 1 describes how each operator handles the barrier and reports its state.

Algorithm 1 Retrieving a state of an operator

```
Input: Interesting Operator $\theta$, Operator $x$, SubDAG $G(\theta)$
1:  $\mathcal{E} \leftarrow \{e \mid e \in x.getInputEdges()$ if $e \in G(\theta)\}$
2:  $B \leftarrow$ null
3:  if $x \neq \theta$ then
4:      while $\mathcal{E}$ is not empty do
5:          $m \leftarrow receiveMessage()$
6:          if $type(m)$ == Barrier then
7:              $B \leftarrow m$
8:              $m.edge.disable()$  ▷ Block messages from this edge.
9:              $\mathcal{E}.remove(m.edge)$
10:         else                    ▷ Receive and process a data message.
11:             $x.process(m)$
12: else
13:     $B \leftarrow generateBarrier()$
14: $reportState(x)$
15: $sendToAllOutputEdges(B)$
16: $enableAllInputEdges()$  ▷ Unblock messages from all edges.
```

Due to the distributed nature of the algorithm, operators along different paths starting from θ in $G(\theta)$ process messages independently, without blocking each other. On each path, the processing time of a message is the sum of the processing times of all its operators. Let K_p represent the number of in-flight messages along path p, and let T_p denote the total processing time per message on that path. Therefore, the total time required to collect operator states along path p is $K_p \times T_p$. Since the messages on paths are processed in parallel, the path with the maximum $K_p \times T_p$ is the bottleneck, and its time is the overall duration of the snapshot-retrieval process.

4 POST-EXECUTION DEBUGGING

In this section, we present a novel time-travel debugging paradigm on dataflow systems to allow users to control and investigate past executions. The paradigm includes two debugging primitives, namely "jump" and "controlled replay."

4.1 Jumping to a Past Interaction

One primitive of time-travel debugging is to roll back the states of the operators to a past interaction. This primitive is especially important when, after several interactions, the user wants to investigate a past interaction. To do so, she can use a Jump instruction to revert the workflow execution to the earlier interaction point and begin debugging from there.

Definition 4.1 (Order of Interactions). We denote the timestamp when a request of interaction I is received by the coordinator as $T(I)$. The *order* of interactions is determined based on these timestamps. Specifically, for any two interactions I_1 and I_2, interaction I_1 occurs before I_2 (denoted $I_1 \prec I_2$) if and only if $T(I_1) < T(I_2)$.

Definition 4.2 (Interaction History). Given an interesting operator o in a workflow, an *interaction history* of an execution is a sequence $H = [I_0, \ldots, I_n]$, where each I_i is an interaction corresponding to a snapshot, and $I_0 \prec I_1 \ldots \prec I_n$. The snapshot retrieved in interaction I_i is denoted as $S(I_i)$. Snapshot I_0 contains the initial states of the operator o and all its downstream operators.

Definition 4.3 (Jump). Given an interaction history H, a *jump* to an interaction $I_k \in H$ is to rollback the operators in the snapshot to their states at I_k. After the jump, all the operators covered by the snapshot are waiting for messages.

Figure 7 shows two interactions I_1 and I_2 in execution of the workflow in the running example. Both snapshots of the interactions are tuple-consistent. At I_2 the user sees that Bob is added to the blacklist, while at I_1 the blacklist is empty. The user jumps to I_1 to start replaying the execution to identify why Bob was blacklisted.

Supporting Jumps. To support a jump instruction, IcedTea needs to capture the state of each operator involved in an interaction during the original execution. Since users can request any number of interactions, storing all these snapshots could introduce significant overhead in storage and time. To reduce the impact of the runtime performance, for each interaction, IcedTea captures a logical timestamp when an operator processes an interaction message. Later, the system utilizes the timestamp to reconstruct its state at an interaction.

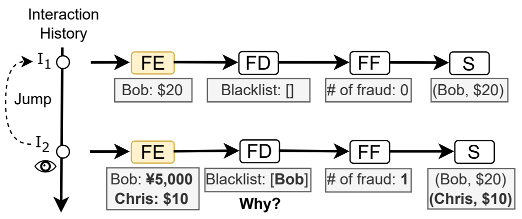


Figure 7: After seeing Bob has been added to the blacklist at I_2, the user wants to rollback the workflow to an earlier interaction point I_1 to start an investigation.

For simplicity, we first assume that an operator's computation is *deterministic*. That is, (1) given the same initial state of an operator and an input message, the operator will always produce the same output messages and reach the same resulting state after processing the message.; and (2) for each execution, every source operator generates the same sequence of output messages. If the computation of an operator is deterministic, by recording logical timestamps, it is necessary to reproduce its state to the timestamp of interest. Otherwise, we need to materialize its input data to ensure we can reconstruct the operator's state from earlier data messages.

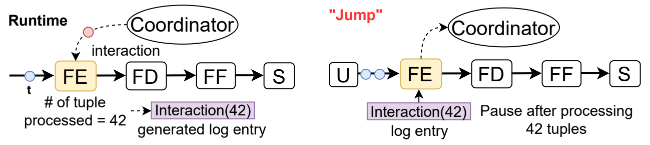
We will relax these two assumptions in Section 5.

Each operator contains a counter to keep track of how many tuples have been processed by the operator. Once it receives a control message for an interaction, it writes its current counter value to a log file. For instance, Figure 8a shows that a user interaction is processed by operator FE. After handling this interaction, the counter shows the operator has processed 42 tuples. The operator writes this number to its log. Each downstream operator of FE also generates a log entry with its respective number of processed tuples after receiving the propagated interaction message.

If the user wants to jump to a previous interaction snapshot, the coordinator sends a control message to restart all the operators. For each operator in the snapshot, IcedTea specifies the target interaction and prompts it to process its tuples. After the operator reaches the recorded number, it sends a control message to the coordinator and awaits future messages from the coordinator. Once the coordinator has received control messages from all the operators in the snapshot, the jump is complete. For operators not included in the snapshot, such as the source operator U, they continue producing tuples because the user is not interested in their operator states.

To achieve a waiting state for operators in $G(\theta)$, we need to "stash" newly incoming tuples, i.e., they are temporarily stored without being processed. There are two ways to do so. One approach is to stash the tuples on the sender side. In this case, the coordinator sends the tuple count for its downstream operator to the sender operator. After generating the target number of tuples, the sending operator can simply pause itself. Another approach is to stash the tuples on the receiver side. That is, once the receiver operator reaches the target state, it starts to stash the data tuples on every input edge without further processing. We call this action a "pause". This approach ensures that the state of the receiver operator remains unchanged after reaching the target state. In this approach, the upstream operators may generate a large number of tuples. We use a backpressure mechanism [16] to manage the flow, stopping the tuple generation from the upstream operator when necessary.

For instance, Figure 8b shows how IcedTea restores the state of the operator FE during the jump to the recorded interaction. The coordinator restarts all the operators, and FE checks its step count after processing each tuple until it reaches the target number 42. Then, it pauses its processing, waits for the next control message from the coordinator, and notifies the coordinator. The same operations happen for all the downstream operators of FE. Since the source operator U is continuously producing tuples, all the tuples are now queued to be processed on the edge between U and FE due to the pause of FE.



(a) Upon an interaction, FE records its counter value 42. **(b) During a jump, FE processes 42 tuples, then pauses.**

Figure 8: Logging an interaction and jumping to this interaction after execution.

4.2 Controlled Replay between Interactions

Another primitive in time-travel debugging involves replaying the execution between interactions. Once the user reverts the execution back to a past interaction, she can replay the computation of the original execution. This primitive is important in helping the user understand how the state of each operator changed in the original execution, step by step. Since the purpose of a workflow is to process data, we focus on how to let users trace the processing of tuples, and correspondingly define step-related concepts based on tuples.

In our previous example, after jumping back to the interaction I_1, the user can replay the execution from I_1 to I_2 by using two instructions on a particular tuple, namely "step-over" and "step-into". After the user jumps back to I_1, FE is waiting for its next tuple, i.e., $(Chris, \$10)$. Since this tuple is irrelevant to Bob's transactions, the user is not interested in how each operator processes this transaction. By doing a "step-over", the user can get a snapshot when a tuple is completely processed. Specifically, we first introduce the concept of a "tuple's scope," which includes the tuples generated during the processing of a tuple.

Definition 4.4 (Tuple scope). Given a workflow W, the *scope* of an input tuple t of an operator o, denoted as $\mathcal{D}(W, o, t)$, is a set of tuples defined as follows:

(1) t is in $\mathcal{D}(W, o, t)$.
(2) For each tuple d in $\mathcal{D}(W, o, t)$, if an operator processes the tuple d and produces zero or more output tuples $\{d'_1, \ldots, d'_n\}$, all the produced tuples are also in $\mathcal{D}(W, o, t)$.

Figure 9 shows the tuple scope of the tuple x, which includes x, x_1, x_2, and x_3.

Definition 4.5 (Step-Over). Given a workflow W, an interesting operator θ, and an input tuple t to an operator in $G(\theta)$, a "step-over"

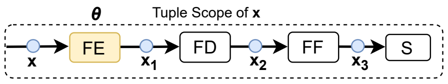


Figure 9: The tuple scope of x is a set of four tuples, which contains x, x_1, x_2, and x_3.

of t results in a snapshot where it reflects all tuples in $\mathcal{D}(W, t)$ have been processed. All the operators in the snapshot wait for messages. For opeators outside the snapshot, they still process tuples.

Supporting step-over. When the user requests a step-over on a tuple, the coordinator sends a control message to the receiving operator and all of its downstream operators, instructing them to continue processing. Once the receiving operator receives the control message, it processes the first input tuple and generates output tuples. It also appends a barrier after those output tuples. Upon receiving all barriers from its upstream, an operator propagates this barrier to its downstream, pauses, and awaits further stepping instructions. Once the receiving operator and all its downstream operators send their states to the coordinator, the step-over instruction is completed.

After jumping back to I_1, the user decides to take a step-over on tuple $(Chris, \$10)$, which is shown in Figure 10. During this step-over, the four operators process tuple $(Chris, \$10)$ and result in another snapshot S_1. In this snapshot, a value of \$10 is added to *FE*'s state and the \$10 transaction is approved.

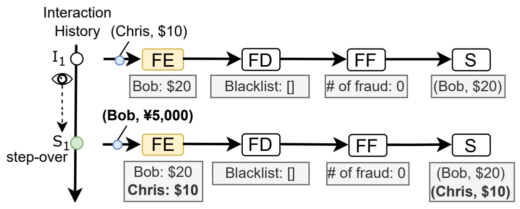


Figure 10: A step-over of the tuple $(Chris, \$10)$ results in a snapshot where the tuple of Chris's payment is processed sequentially by the three operators.

After inspecting the snapshot S_1, the user sees a pending tuple $(Bob, ¥5,000)$ to be processed by FE. She wants to take a close look at how each operator processes this tuple. In this case, she can use the "step-into" debugging instruction.

Definition 4.6 (Step-Into and Step-Out). Given a workflow W, an interesting operator θ, and an input tuple t to an operator o in $G(\theta)$, a "step-into" of t results in a snapshot where it reflects tuple t has been processed by o. All the operators in the snapshot are waiting for messages. A corresponding "step-out" operation results in a snapshot where all the tuples in $\mathcal{D}(W, t)$ are processed.

Supporting step-into and step-out. If the user selects the input tuple and requests a step-into, the coordinator sends a control message to the receiving operator of the tuple to instruct the operator to process the tuple. The processing results in a state change of the operator and the generation of output tuples. The operator also sends a barrier after its output tuples to its downstream. The updated state is sent to the coordinator through a control message. When downstream operators receive these output tuples, they will also notify the coordinator that they have pending input tuples.

Suppose a step-into is applied on the operator o. IcedTea allows the user to take a step-out by sending control messages to the downstream operators of o. After receiving this control message, each of them continues its tuple processing until it processes the barrier. After that, it again pauses itself and sends its state to the coordinator.

The user decides to take a step-into on tuple $(Bob, ¥5,000)$ to inspect how FE processes it. During this step-into, FE incorrectly parses the amount of the payment. The currency unit for the payment is changed from ¥ to \$ in the output tuple. This step-into results in another snapshot S_2. After that, the user takes a step-out operation. All operators continue to process Bob's payment. As a result, Bob is added to the blacklist, and the fraud count of FF increases.

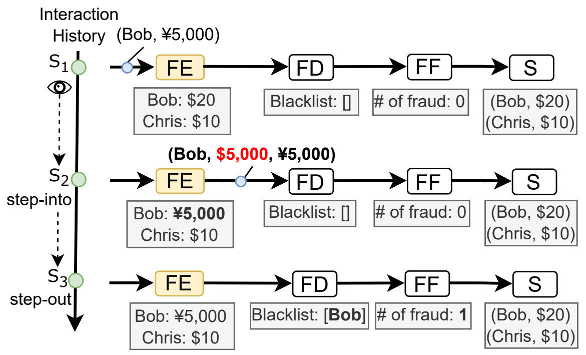


Figure 11: A step-into of the tuple $(Bob, ¥5,000)$, resulting in an output tuple with the payment value changed to \$5,000. After a step-out, the resulting snapshot shows that Bob has been added to the blacklist, and a fraud transaction has been filtered by FF.

5 GENERALIZATIONS

In previous sections, we made a few assumptions: (1) The workflow is a chain of operators; (2) The computation of each operator is deterministic. In this section, we relax these assumptions and generalize the results. We first relax the assumptions on the input edges and output edges of an operator, then consider the case where the computation of an operator is not deterministic.

5.1 Operators with Multiple Input Edges

We consider DAGs where some of the operators have more than one input edge. The order in which tuples arrive at the operator could be non-deterministic in different executions, and the state of the operator can depend on the tuples' arrival order. To make sure the operator can be determinstically replayed, IcedTea captures the arrival order of the input tuples in the original execution. Specifically, IcedTea writes additional *Input* log records, each of which the edge of an input tuple. During post-execution debugging, the operator processes input tuples from the incoming edges in the logged order.

Note that sometimes the interesting operator also needs to process tuples outside the current tuple scope. Suppose we have an operator C processes tuples from both operators A and B. This means a tuple from B can be processed by C, between the processing of tuples from A. If the user marks operator A as the interesting operator, tuples from B are automatically consumed during the stepping instructions to repeat the exact execution.

5.2 Operators with Multiple Output Edges

For an operator with multiple output edges, it can produce tuples to each edge for an input tuple. To support the stepping instructions, the operator can send any barrier created during these instructions to all its output edges. Due to the distributed nature of the execution, the output tuples are processed by the downstream operators in no particular global order. During the post-execution debugging, the user can take stepping instructions on any of these tuples. As an example, Figure 12 shows a tree-shaped workflow with an operator B that has three output edges. Suppose we want to do a step-into on this operator for a tuple t. After processing the tuple, the operator outputs three tuples, along with three barriers. After all three downstream operators receive their corresponding tuple, the coordinator prompts the user for step instructions. The user can choose any of the three tuples to perform a step instruction.

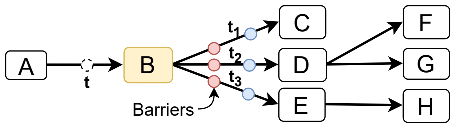


Figure 12: An example DAG with operators having more than one output edge. After a step-into on t, the user can take step instructions on t_1, t_2, or t_3.

5.3 General DAGs

A general DAG can have operators with multiple input edges as well as operators with mulitple output edges, which can result in more than one path between two operators. As a result, the tuples generated by two tuples of an upstream operator could arrive at a downstream operator in an order different from the order in which they arrived at the upstream operator. Consider an example in Figure 13. Operator A processes tuple x then tuple y. Tuple x_2 in the scope of x is produced by operator B, and tuple y_2 in the scope of y is generated by operator C. At operator D, the processing order is different, as y_2 is processed *before* x_2. In other words, the processing of scopes of tuples x and y can overlap in the temporal dimension. In this case, a step-over on tuple x is not feasible because operator D must process a tuple within the scope of y before it can process x_2, which requires that tuple y to be processed first by operator A.

To detect such a case, IcedTea requires operators to write additional *Output* log records, which include the number of output tuples generated on each downstream edge for an input tuple. Using these log records, IcedTea can trace the scope of each tuple and the sequence in which operators handle these scopes. It can then determine if the processing of two tuple scopes overlaps. For an input tuple in such a case, its step-over is not applicable. Figure 14 illustrates an example for the workflow in Figure 13. For tuple x, operator A records a log entry $Output([(B,1)])$. The recorded $(B,1)$ indicates that the operator sends one tuple to operator B. Subsequently, B forwards this output tuple to D. The log for operator D shows that it first processes a tuple from operator C before handling the one from operator B. As the log does not show any output tuple generated from C, the processing of the scope of x overlaps with another tuple's scope. So a step-over on tuple x is not feasible.

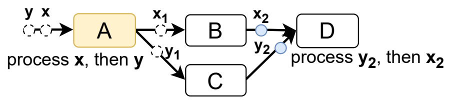


Figure 13: A step-over on x requires A to also process y since D needs to process y_2, which is in the scope of y, before it processes x_2.

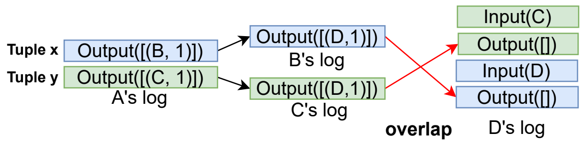


Figure 14: The log records from operators A to D show that the processing of the scope of x overlaps with the processing of the scope of y.

5.4 Non-deterministic Operators

So far, we assume each computation step of an operator is deterministic. We now relax this assumption. As an example, consider a sentiment analysis operator that processes an input tuple by generating a probability distribution for possible subsequent tokens. This step is deterministic. Then the operator calls a function `randInt` to randomly select a token from the distribution. This `randInt` function produces non-deterministic results. Calling external services (e.g., OpenAI's ChatGPT service [8]) may also make the operator non-deterministic. To handle non-deterministic computation in a replay process, particularly in user-defined functions (UDFs), IcedTea provides an API called `logResult()` for users to wrap any function calls in their custom operator logic. The user is responsible for wrapping their non-deterministic function calls using this API. We log the result value of each function call from the original execution. For instance, the sentiment analyzer calls `logResult(randInt)` to denote that `randInt` is non-deterministic, and its result should be logged. During the original execution, the `logResult()` function executes the supplied function as usual and then saves the returned value in the log record. In each replay run, `logResult()` skips invoking the original function and retrieves the saved returned values directly from the log records so that we can ensure the returned value is deterministic.

5.5 Large Operator States

During the replay process, some operators may have a large state, which could cause significant overhead on storage and network when it is transferred to the user. To reduce this overhead, we can use data structures such as Merkle trees [5] to incrementally calculate the updates between two consecutive states. For instance, operators with their state as a hash map (e.g., GroupBy or HashJoin) can save only the updated key-value pairs between states. Similar approaches can also be used to efficiently store checkpoints.

6 SUPPORTING RESPONSIVE DEBUGGING

Since time-travel debugging is a user-facing experience, it is crucial to support requests responsively. In this section we consider how to use checkpoints to reduce latency, and discuss ways to provide upstream data for an interesting operator. We then study how to do checkpointing judiciously to meet a responsiveness requirement.

6.1 Reducing Latency Using Checkpoints

We can support a jump instruction by replaying the log records from the initial state. This process is time-consuming, and we can reduce the time by doing checkpoints at some of the interactions. A *checkpoint* at an interaction is a saved copy of the snapshot at the interaction, which includes the states of the operators in $G(\theta)$. Suppose the workflow state of an interaction I_j is checkpointed. To jump back to the state at I_j after the original execution, we can simply retrieve the state of each operator from the checkpoint. Suppose the user wants to jump to a non-checkpointed interaction I_k after I_j. We can load the checkpointed snapshot of I_j and let each operator replay its execution until it reaches I_k. Replaying the log after I_j significantly reduces the latency compared to replaying the log from the initial state.

To reduce the time to provide data records to θ that were consumed after the checkpointed interaction, operator θ can store all its input tuples after the interaction, which can be read after a jump instruction to support replay. Alternatively, we can let the upstream operators of $G(\theta)$ repeat their computation from their initial states to re-generate the tuples to $G(\theta)$, which uses its log to decide the new tuples to process. To reduce this data-regeneration time, we could also checkpoint the upstream operators and start regenerating their tuples from their previous checkpoint.

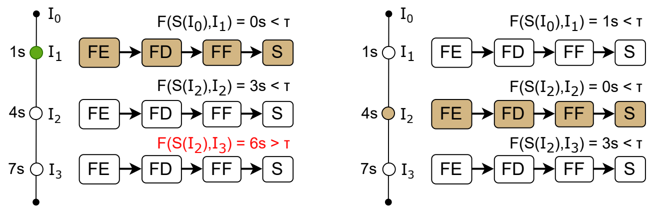
6.2 Selective Checkpointing with Responsiveness Guarantee

Assuming a tuple can be processed by the workflow quickly, each step instruction can be served responsively. Next, we focus on the responsiveness of jump requests. We assume a time limit τ that represents the maximum amount of time a user is willing to wait for each jump instruction. Ideally, each request should be served within τ. We study the following optimization problem: given τ, select a set of interactions to the checkpoint so that after execution, the user can jump to each interaction within τ. The latency of the jump depends on several factors. First, the time to load saved checkpoints depends on whether the checkpoints are saved in memory or on disk. Second, during the replay process, operators must wait for messages from an input edge to enforce the message-processing order, which could impact the time.

We denote the time to jump to an interaction I_k from a checkpoint of I_j as $F(S(I_j), I_k)$. For simplicity, we focus on the case where states are saved in local memory at operators and the input data for $G(\theta)$ is available. In this scenario, we assume the amount of time for a jump to the target interaction from a checkpointed interaction is equivalent to the time gap between them, i.e., $F(S(I_j), I_k) = T(I_k) - T(I_j)$. This assumption will be verified in our experiments. The results can be extended to checkpoints saved on disk or at the coordinator, with additional time based on checkpoint size and network cost.

Definition 6.1 (Checkpoint Plan). Given an interaction history $H = [I_0, \ldots, I_n]$, a *checkpoint plan* is a subset R of H, where each $I_k \in R$ is checkpointed.

Given a checkpoint plan, a jump instruction of interaction I_k starts from the latest checkpoint before I_k. Note that different checkpoint plans have different storage costs, and not every checkpoint plan can meet the responsiveness requirement of a user. Figure 15 illustrates four interactions $I_0, \ldots, I_3$, with I_1 as the only checkpointed interaction. Assume the time threshold $\tau = 5$ seconds. If the user requests a jump to I_3, the replay process starts from the state of I_1 and takes seven seconds, which is more than τ, so this plan does not satisfy the responsiveness requirement. Figure 15b illustrates another checkpoint plan that checkpoints I_2. This plan meets the responsiveness requirement for any jump request.



(a) A checkpoint plan including I_1 cannot support a jump to I_3 within time limit $\tau = 5s$.

(b) Another plan including I_2 supports a responsive jump to any interaction.

Figure 15: Two checkpoint plans.

Next we present two approaches to selecting a set of interactions to checkpoint.

Online Approach. During the original execution of the workflow, we make checkpoint decisions each time the user makes an interaction. This approach makes online decisions to checkpoint a subset of interactions. For each interaction, the coordinator first checks if the replay of the current interaction can be completed from the latest checkpoint within τ. If so, we do not checkpoint the current interaction. Otherwise, we do a checkpoint. While this approach makes all the decisions in-place of the original execution, it may introduce large storage overhead as the checkpoint points may be more than needed.

Offline Approach. Another approach is to make checkpoint decisions after the execution. During the original execution, we estimate the checkpoint cost at each interaction without doing any checkpoint. This execution can also be viewed as a "profiling phase." After the execution, we use collected information to select a set of interactions to do checkpoints, aiming to minimize the total storage size. We then restart the execution with logs, and checkpoint the selected interactions, which is referred as a "checkpointing phase".

After that, each jump instruction can be handled using the checkpoints responsively in the debugging phase. This approach has two advantages. First, if the user does not want to investigate the past interactions after the execution, especially if the execution finishes successfully, we do not need to introduce any checkpoint overhead during the execution. Second, if the user does want debugging, we can use the cost information to checkpoint the interactions with a minimal cost while satisfying the responsiveness requirements.

A natural question is how to select an optimal set of interactions for checkpointing after the profiling phase. Consider an interaction history $H = I_0, \ldots, I_n$, and a function $C(S(I_k))$ that measures the cost of saving the workflow state $S(I_k)$ at interaction I_k. For each subset H' of H, we have a checkpoint plan in which each interaction in H' is checkpointed. The cost of this plan is the summation of the checkpoint costs for the interactions in H'. We develop a dynamic programming algorithm to solve this problem. It starts by initializing an empty set for an empty interaction history. For each sub-history, it evaluates possible checkpoint positions, and checks if they allow jumps within τ. Feasible checkpoints are combined with previous optimal sets to form candidate plans. The algorithm selects the plan with the minimal total checkpoint cost and updates the optimal set accordingly. This process continues until the optimal checkpoint set for the entire interaction history is determined.

7 EXPERIMENTS

In this section, we present our experimental results to evaluate the proposed IcedTea system.

7.1 Settings

Datasets. We used three datasets shown in Table 1. Dataset 1 was generated using the TPC-H benchmark [32] with a scale factor of 1. Dataset 2 had 1M tweets sampled from November 2019 to April 2020, with 34 attributes, including user id, location, content, id, and creation time of each tweet. Dataset 3 was a collection of 945K Amazon reviews [26] in two categories called "All-Beauty" and "Luxury-Beauty." Each record contained the content, rating, reviewer's name, and time of a review.

Dataset	Name	Table	Field #	Tuple #	Size (MB)
1	TPC-H	lineitem	16	6M	772
		orders	9	1.5M	173
		customer	8	150K	24.3
2	Twitter	tweets	34	1M	1140
3	Reviews	all-beauty	14	371K	120.8
		luxury-beauty	14	574K	192.9

Table 1: Datasets used in the experiments.

Workflows. We created five workflows shown in Figure 16 to cover three common types of dataflows. (1) *Long running workflows*: they included workflow W_1 based on TPC-H query 1 and W_2 based on TPC-H query 10, both on the TPC-H dataset. (2) *Workflow with multiple parallel branches*: this included workflow W_3 that analyzed the Twitter dataset related to climate change. (3) *Workflows with user-defined functions (UDFs)*: they included W_4 and W_5. Workflow W_4 analyzed the Twitter dataset to produce a monthly aggregation of tweets containing the keyword `happy` or `glad`. Workflow W_5 analyzed the reviews to compute the sentiment of individual reviews and aggregate the count by each sentiment, corresponding to our running example. The Sentiment Analyzer operator uses a k-nearest-neighbor (KNN) model and is sensitive to the arrival order of the records. The interesting operator for each workflow is marked in yellow. Each operator kept track of the number of input tuples processed from each upstream edge in its internal state.

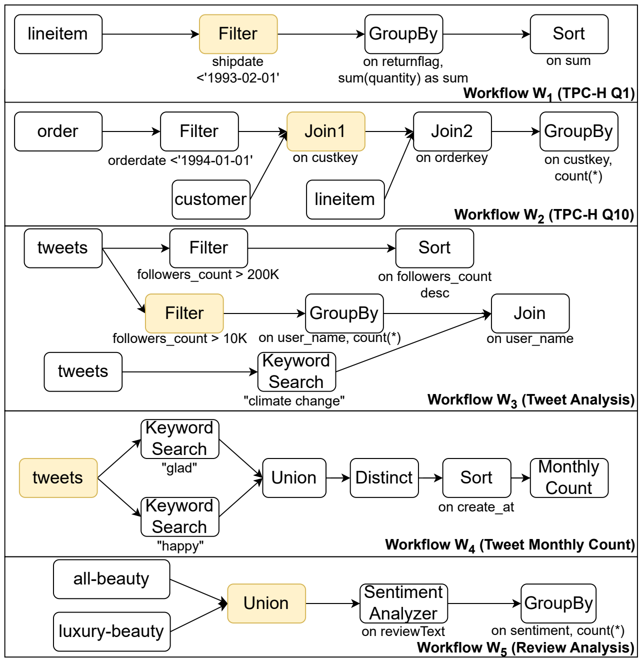


Figure 16: Workflows used in the experiments.

Implementation of IcedTea**.** We implemented IcedTea on top of Texera [35], a GUI-based collaborative data science workflow system. Texera is powered by the Amber engine [23], which supports forward debugging. In this implementation of IcedTea, each operator's log records were written by a separate thread, enabling concurrent logging and message processing. The log records and saved states were stored in memory. During a jump, we let the receiver operators stash the data. To accelerate jumps, we adopt the data regeneration approach and extend checkpointing to all operators in the workflow. Specifically, for operators upstream of $G(\theta)$, which do not require tuple consistency, the coordinator sends control messages to initiate the Chandy-Lamport algorithm; the inflight messages are recorded on the receiver side. For operators downstream of $G(\theta)$, the algorithm described in Section 3.3 is applied after initiating the Chandy-Lamport algorithm. We summed the byte-sizes of each operator's serialized state as the cost.

Environment. The experiments were conducted on a virtual machine (VM) hosted on the Google Cloud Platform. This VM used the e2-highmem-4 type, with a 100GB SSD persistent disk and Debian 5.10.191 as the operating system. We ran each workflow three times in each experiment and averaged their results. We simulated interactions by periodically sending a control message to the interesting operator of each workflow.

7.2 Stateful operators in workflows

To determine if stateful operators are common in workflows, we analyzed three sets of workflows from different streaming systems: (1) Flink-streaming [13], (2) Texera [35], and (3) Kafka [21]. Systems with a stage-by-stage execution model, such as Spark, were excluded from our analysis, as IcedTea requires a pipelined execution model. We specifically looked for operators such as join, aggregate, or custom UDFs that manipulate user-defined states, as these are common stateful operators. As shown in Table 2, at least 34% of the workflows contained stateful operators.

Dataset	Total Workflows	Stateful Workflows	Percentage
1	24	10	41.7%
2	1,349	463	34.3%
3	14	12	85.7%

Table 2: Analysis of workflows containing stateful operators.

7.3 Comparing IcedTea and forward debuggers

We conducted a user study to compare the effectiveness of IcedTea and the existing forward debugger in Texera. We invited six participants to debug five workflows, labeled Q_1 to Q_5, each containing bugs. The bugs in Q_1, Q_2, and Q_3 were caused by corrupted input tuples, which led to corrupted operator states and unexpected behaviors. The bugs in Q_4 and Q_5 were caused by bad tuples that triggered runtime errors but did not affect the operator state. The participants were given the task to identify the root cause of the bugs and pinpoint the specific tuple that revealed the bug. We divided the participants into two groups, A and B, each consisting of three members. Group A used the forward debugger, which allowed them to pause and resume the execution, and use print statements to inspect the system's state. Group B used IcedTea, with the ability to roll back to previous system states and step forward to observe changes in the state. We measured the total time taken by each participant to debug a workflow, from the time the execution started to the time they identified the problematic tuple.

Workflow	Stateful	Execution progress before the bug	Forward debugger (seconds)	IcedTea (seconds)
Q_1	Y	16%	440	407
Q_2	Y	38%	998	600
Q_3	Y	2%	563	541
Q4	N	18%	286	690
Q_5	N	73%	1,020	467

Table 3: Average debugging time (in seconds) for Q_1 to Q_5.

Table 3 shows the results, including whether the buggy operator of a workflow was stateful or not, the progress of the execution before each bug was observed, and the participant's time to find the bug. For Q_1 to Q_3, IcedTea was more effective because its rollback feature allowed users to trace and revert the operator state, which is beneficial for stateful operators. In Q_4, its buggy operator was stateless and the bug appeared after 18% of the execution. The participants preferred to rerun the workflow from scratch, and were able to find the bug faster. For Q_5, the bug occurred after 73% of the execution, IcedTea's rollback feature saved the time of the participants by avoiding the overhead of rerunning the entire workflow from the beginning.

7.4 Runtime Overhead in Original Executions

To ensure a minimal impact of IcedTea on executions, we evaluated the runtime overhead for supporting time-travel during the original workflow execution. We ran all five workflows with and without the support for time-travel debugging, and compared their corresponding total runtime and space usage. In this set of experiments, we did not enable the checkpoint techniques. For the runs with time-travel debugging enabled, we generated an interaction every 10 seconds to simulate an interaction-intensive scenario.

Time. As shown in Figure 17, after enabling time-travel debugging, the total execution time remained comparable to the original execution without time-travel debugging. The overhead introduced by logging was less than 2% of the total execution time across all the workflows. It shows the low overhead of recording only the input order and user interactions. For W_1, W_2, W_3, and W_5, enabling time-travel debugging added 3-7 seconds of overhead on top of hundreds of seconds of the original execution time. The workflow W_4 had fewer concurrent operators, and its overhead was within 1 second.

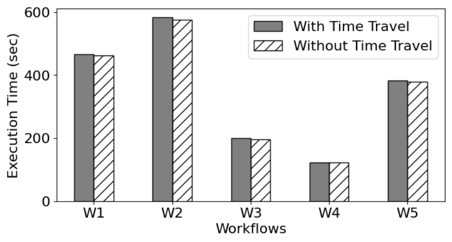


Figure 17: Time overhead of time-travel.

Space. We measured the size of log records generated for jump and steps to evaluate the space overhead of time-travel. As shown in Figure 18, the log content generated by IcedTea was minimal compared to the size of the input data. Note that the input data was not materialized because it was loaded from a stored dataset. For each workflow, the total log size was less than 2% of the data size. It also shows the distribution between different log record types. For workflow W_1, all records were interaction logs since the workflow formed a chain of operators. As a result, no $Input$ or $Output$ logs were needed, leading to 1.19 MB log size. For workflow W_2, $Join1$ and $Join2$ needed $Input$ logs to track tuple order, which resulted in 3.82 MB log size. Workflow W_3 had fewer tuples sent to the $Join$ operator compared to W_2, which reduced the number of $Input$ log records and the log size was 2.14 MB. For workflow W_4, the two paths between the interesting operator and the $Union$ required all operators to retain $Output$ log records, while only the $Union$ kept $Input$ records. This led to a log size of around 18 MB, largely due to the $Output$ logs. For workflow W_5, the total log size was 3.44 MB. Notice the log records could be easily compressed using techniques such as variable-length encoding and dictionary encoding. In addition, for unary operators, storing identifier information for their unique input edge is unnecessary.

Interaction Frequency. We measured the impact of frequent interactions on IcedTea, beginning with a baseline of one interaction

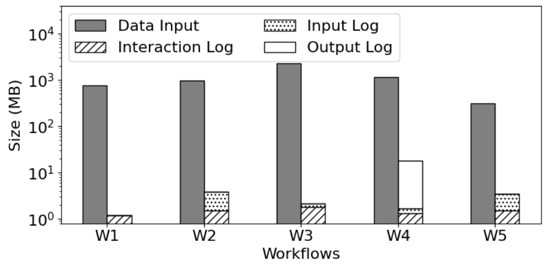


Figure 18: Space overhead of time-travel.

every 10 seconds and gradually increasing the frequency to 10 interactions per second. To better simulate real-world scenarios, we capped the experiment at 10 interactions per second, which exceeds typical human interaction rates. As shown in Figure 19, the additional execution time overhead remained below 5% for W_1 to W_3, and under 12% for W_4 and W_5. The results show that IcedTea can handle highly frequent interactions efficiently.

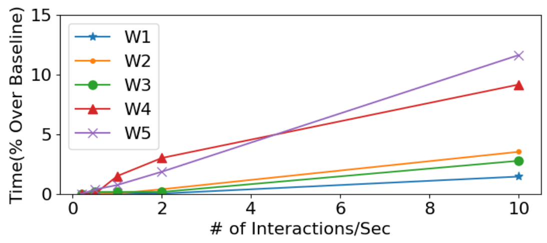


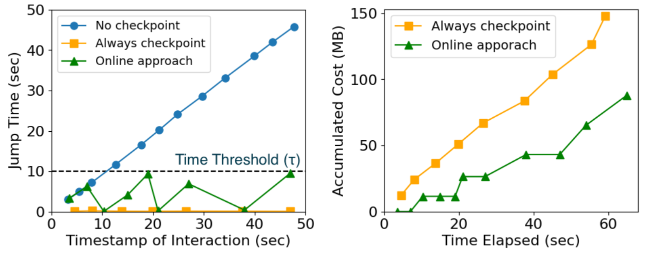
Figure 19: Impact of interaction frequency on execution.

7.5 Evaluating Jump Instructions

To evaluate the effect of taking checkpoints to accelerate jumps, we conducted an experiment using three different checkpoint strategies. The first strategy was checkpointing none of the interactions. The second strategy was checkpoint all interactions. The third strategy corresponded to the online checkpointing approach. We triggered interactions every 5 seconds during the execution. The time threshold for a single jump request was $\tau = 10$ seconds. The results for the workflows were similar. Due to limited space, we showed the results of W_3.

Figure 20a shows the jump latency of W_3. For jumps without checkpoints, the latency rapidly increased as the target-interaction time increased. The latency surpassed the τ threshold for all the requests after 12 seconds from the beginning of the execution. We had the following two observations: (1) The jump latency increased as the time gap between I_0 and the target interaction increased, since operators had to process more tuples. (2) Each jump latency was very close to the corresponding part of the original execution time. These observations highlighted the importance of responsive jumps, as users do not want long wait times.

Both the second and third strategies achieved the responsiveness requirement. For the second strategy, the system loaded the state from memory, which had a sub-second latency. For the third strategy, the delay varied from sub-seconds to 9.5 seconds. Figure 20b presents the cumulative checkpoint cost for each approach over time. The second strategy incurred a significant total checkpoint cost. In contrast, the online approach generated checkpoints for some of the interactions, resulting in a lower cost.



(a) jump time under different checkpoint strategies. **(b) Accumulated checkpoint cost over time.**

Figure 20: Jump latency and checkpoint cost under three different checkpoint strategies for workflow W_3.

Effect of threshold τ on checkpoint cost. To evaluate the impact of τ on the performance of online checkpointing, we varied its value in the range of 0.5 to 20 seconds. For each workflow, we generated multiple interactions, with an average gap of 3 seconds. We used online approach with each τ value. When τ was small (e.g., 0.5 seconds), this online approach created a checkpoint for almost every interaction. The associated cost could be viewed as the maximum total cost needed for time travel. We then incrementally increased the value of τ to evaluate how it affected the overall checkpoint cost.

Figure 21 showed the results. The effect of the τ value remained consistent across all the workflows. When $\tau = 5$ seconds, the overall checkpoint cost is reduced to about 40% compared to the 0.5-second setting. Changing τ to 10 and 20 seconds further decreased the cost to roughly 20% and 10%, respectively. For workflow W_5, there was a small cost difference between the 10 and 20-second settings. This result was due to the frequent fluctuations in the workflow's state size during the execution of W_5. These fluctuations caused the online approach to generate checkpoints with a high cost.

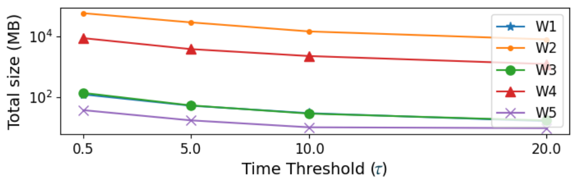


Figure 21: Checkpoint costs for various τ values relative to the cost of checkpointing every interaction. W_1 and W_3 had similar results.

Comparing online/offline checkpointing. To evaluate the benefits of the offline method that selects an optimal subset of interactions to checkpoint, we compared the total checkpoint cost between the result from the online approach and the offline DP algorithm. We used two values for τ: 5 seconds and 60 seconds. We randomly triggered multiple interactions to generate an interaction history.

We ran both algorithms given the same interaction history and measured their total checkpoint cost. Then we computed the percentage of cost reduction of the optimal checkpoint plan over the online checkpoint plan. Figure 22 shows the results.

When we set τ to 5 seconds, the optimal checkpoint plan had a cost 10% less than the cost of the online approach. It was because, for this τ, the latter created frequent checkpoints, leaving less improvement potential. As τ was 60 seconds, the potential for cost reduction increased. For workflows with a stable state size, such as W_1 and W_2, the online checkpoint plan was nearly as effective as the optimal one, showing a difference of less than 10%. Similarly, in workflows with a state size that did not have too many fluctuations, such as W_4, the benefits of the optimal plan were limited. For workflows with frequent state-size fluctuations, such as W_3 and W_5, the optimal plan reduced the cost up by 40%.

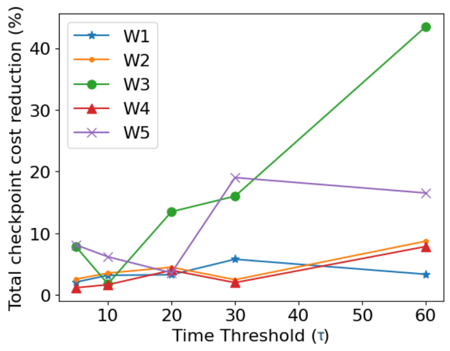
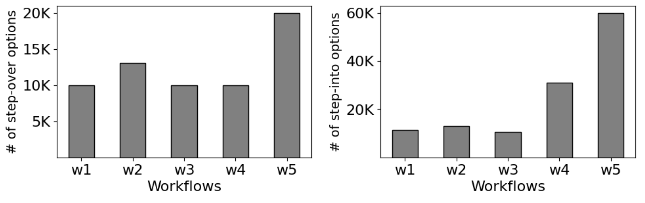


Figure 22: Cost reduction of offline checkpointing compared to online checkpointing.

7.6 Evaluating Step Instructions

We evaluated the granularity of user control in the debugging primitives by examining the number of step options a user could take in an execution. Recall that for each input tuple of an operator, IcedTea allows the user to choose from debugging options such as step-over, step-into, and step-out. Given the shape of a dataflow, for a particular tuple, it may not have step-over when the operator has multiple paths to a downstream operator that causes overlap. Also, the depth of each tuple scope can influence how deep one can step-into a scope. In this experiment, we ran workflows from W_1 to W_5 with 10,000 tuples ingested into their source operator and counted the number of step options available to the user.

Step-over. For each workflow execution, we counted the total number of step-over options for the tuples at the respective interesting operator. As shown in Figure 23a, W_1 and W_3 had exactly 10,000 step-over options for the input tuples because the interesting operator is right after the source operator, each input tuple of the interesting operator has a step-over option that user can choose. For W_2, the interesting operator *Join*1 had two upstream sources, and it received about 3K tuples from the *Filter* operator, and 10K tuples from the *customer* source operator. It had around 13K step-over options. In workflow W_4, the presence of multiple paths in $G(\theta)$ led to overlapping tuple scopes, as stated in Section 5.3. As a result, 18 tuples did not have step-over options and the total number of step-over options is 9,982. For workflow W_5, the interesting operator received tuples from both source operators, resulting in 20,000 step-over options.



(a) Total number of step-over's. (b) Total number of step-into's.

Figure 23: Control granularity in terms of the number of step-into options and step-over options.

We evaluated the effect of overlapping tuple scopes. We used workflow W_4 that contained multiple paths between two operators, which can cause overlapping tuple scopes. We ran the workflow with different combinations of keywords for the keyword-search operators to examine the frequencies of overlapping tuple scopes. As shown in Table 4, the number of tuples without step-over options varied as we changed the keywords. For the keywords happy and `glad`, 65 tuples did not have step-over options. For the keywords happy and `birthday`, the number increased. Out of 1 million tweets, 3,152 tuples had their scopes overlapping with other scopes.

Keywords	Number of tuples without step-over options	Ratio over input tuples
`happy` + `glad`	65	0.01%
`happy` + `new year`	2,359	0.20%
`happy` + `birthday`	3,152	0.30%

Table 4: Effect of having multiple paths (workflow W_4)

Step-into. We evaluated the total number of step-into options a user could take for each workflow. In this experiment, we again ingested 10,000 tuples for each source operator. For each workflow execution, we counted the total number of step-into options for tuples of operators. For each tuple, an operator could generate output tuples, and the user can step into each of them. The number of step-into options corresponded to the number of output tuples for each operator. As shown in Figure 23b, for workflows W_1 to W_3, the number of output tuples generated by the operators after the interesting operator was small due to the small number of output tuples of filters and joins. Consequently, the number of step-into options for these three workflows was also small. Workflow W_4 had around 31K step-into options because the sub-DAG starting from the interesting operator covered all the remaining operators in the workflow, causing each output tuple to generate a step-into option. For workflow W_5, none of the operators filtered out input tuples. Thus, each input tuple had exactly one output tuple. The three operators in $G(\theta)$ resulted in 60,000 step-into options.

7.7 Evaluating Large Opeartor States

In this experiment, we evaluated the network cost of sending large operator states using workflow W_3's Aggregate operator after processing 500K tuples, which generated a large state. To further increase the state size, we modified the Aggregate condition to do a group by on "text" instead of "user_name," so that each tweet created a unique entry in Aggregate's state. This adjustment resulted in a Aggregate state size of around 47MB. After 100 step-overs, the total data size became approximately 4.3GB. To reduce this overhead, we applied both a Merkle-tree-based reduction technique with a 4KB block size and an optimization that sends only the updated key-value pairs for incremental updates between step-overs. As shown in Figure 24, the Merkle-tree-based method decreased the

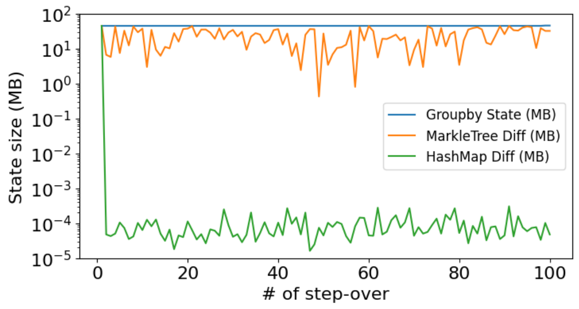


Figure 24: Cost of sending large operator states.

cost of sending each state to about 10MB (a reduction of 47%). By sending only the updated key-value pairs, each step-over transmitted just hundreds of bytes, further reducing the total cost by 99%. These results showed that incremental updates can significantly reduce network overhead for large operator states in IcedTea.

Summary: (1) Many real-world workflows are stateful and IcedTea can reduce the debugging time for state-related bugs. (2) IcedTea had a low runtime overhead in both time and space. (3) Both online and offline checkpointing reduced the time for jump operations. (4) IcedTea provided fine-grained control over the replayed execution using step-over and step-intos. (5) The overlapping of tuple scopes appeared frequently if there were multiple paths between two operators in $G(\theta)$. (6) Existing compression techniques can be used to optimize the overhead of sending large states to the user.

8 CONCLUSIONS

In this paper, we developed a novel system called IcedTea that supports powerful time-travel debugging in dataflows. The system allows users to interact with a distributed dataflow execution to retrieve its global state. After the execution, the user can roll back the dataflow state to any of the past interactions. Using step instructions, they can trace and repeat the past execution to understand how data was processed. We provided a complete specification of this powerful paradigm, presented methods to minimize its runtime overhead, and developed techniques to support responsive debugging instructions. We conducted a thorough experimental evaluation on real-world datasets and dataflows to show that IcedTea can facilitate responsive time-travel debugging with low time and space overhead.

ACKNOWLEDGMENTS

We thank Yunyan Ding, Xinyuan Lin, and Xiaozhen Liu for their invaluable feedback. This work is funded by the National Science Foundation (NSF) under award III-2107150.

REFERENCES

[1] APACHE Spark [n.d.]. Apache Spark, http://spark.apache.org.
[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1383–1394. https://doi.org/10.1145/2723372.2742797
[3] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. 2015. Lightweight Asynchronous Snapshots for Distributed Dataflows. *CoRR* abs/1506.08603 (2015). arXiv:1506.08603 http://arxiv.org/abs/1506.08603
[4] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: The Evolution of Stream Processing Systems. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2651–2658. https://doi.org/10.1145/3318464.3383131
[5] Barbara Carminati. 2009. Merkle Trees. In *Encyclopedia of Database Systems*, Ling Liu and M. Tamer Özsu (Eds.). Springer US, 1714–1715. https://doi.org/10.1007/978-0-387-39940-9_1492
[6] Chengliang Chai, Jiayi Wang, Yuyu Luo, Zeping Niu, and Guoliang Li. 2023. Data Management for Machine Learning: A Survey. *IEEE Trans. Knowl. Data Eng.* 35, 5 (2023), 4646–4667. https://doi.org/10.1109/TKDE.2022.3148237
[7] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75. https://doi.org/10.1145/214451.214456
[8] chatGPT [n.d.]. https://openai.com/blog/introducing-chatgpt-and-whisper-apis.
[9] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. 2015. Deterministic Replay: A Survey. *ACM Comput. Surv.* 48, 2 (2015), 17:1–17:47. https://doi.org/10.1145/2790077
[10] 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
[11] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 137–150. http://www.usenix.org/events/osdi04/tech/dean.html
[12] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3 (2002), 375–408. https://doi.org/10.1145/568522.568525
[13] Flink streaming examples [n.d.]. Stream Processing with Apache Flink - Scala Examples, https://github.com/streaming-with-flink/examples-scala.
[14] GdbRecordReplayDoc [n.d.]. Process Record and Replay (Debugging with GDB), https://sourceware.org/gdb/download/onlinedocs/gdb/Process-Record-and-Replay.html.
[15] Jonathan Goldstein, Ahmed S. Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Peshawaria, Tal Zaccai, and Irene Zhang. 2020. A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications. *Proc. VLDB Endow.* 13, 5 (2020), 588–601. https://doi.org/10.14778/3377369.3377370
[16] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. 2022. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association, 779–805. https://www.usenix.org/conference/nsdi22/presentation/goyal
[17] Muhammad Ali Gulzar, Matteo Interlandi, Tyson Condie, and Miryung Kim. 2016. BigDebug: interactive debugger for big data analytics in Apache Spark. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 1033–1037. https://doi.org/10.1145/2950290.2983930
[18] Muhammad Ali Gulzar and Miryung Kim. 2021. OptDebug: Fault-Inducing Operation Isolation for Dataflow Applications. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia

Koutrika, and Ravi Netravali (Eds.). ACM, 359–372. https://doi.org/10.1145/3472883.3487016

[19] 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

[20] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd D. Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *Proc. VLDB Endow.* 9, 3 (2015), 216–227. https://doi.org/10.14778/2850583.2850595

[21] Kafka streaming examples [n.d.]. Kafka Streams Examples, https://github.com/confluentinc/kafka-streams-examples.

[22] M. A. Klimushenkova and P. M. Dovgalyuk. 2017. Improving the performance of reverse debugging. *Program. Comput. Softw.* 43, 1 (2017), 60–66. https://doi.org/10.1134/S0361768817010042

[23] 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

[24] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, Dahlia Malkhi (Ed.). ACM, 179–196. https://doi.org/10.1145/3335772.3335934

[25] 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

[26] Jianmo Ni, Jiacheng Li, and Julian J. McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 188–197. https://doi.org/10.18653/v1/D19-1018

[27] Douglas Z. Pan and Mark A. Linton. 1988. Supporting Reverse Execution of Parallel Programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, University of Wisconsin, Madison, Wisconsin, USA, May 5-6, 1988*, Richard L. Wexelblat (Ed.). ACM, 124–129. https://doi.org/10.1145/68210.69227

[28] ReplayDebuggerWebsite [n.d.]. Replay - The time-travel debugger from the future, https://www.replay.io.

[29] Astrid Rheinländer, Ulf Leser, and Goetz Graefe. 2017. Optimization of Complex Dataflows with User-Defined Functions. *ACM Comput. Surv.* 50, 3 (2017), 38:1–38:39. https://doi.org/10.1145/3078752

[30] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun C. Murthy, and Carlo Curino. 2015. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1357–1369. https://doi.org/10.1145/2723372.2742790

[31] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proc. VLDB Endow.* 15, 5 (2022), 1119–1131. https://www.vldb.org/pvldb/vol15/p1119-sichert.pdf

[32] tpch [n.d.]. TPC-H Website, http://www.tpc.org/tpch/.

[33] UndoDBWebsite [n.d.]. UDB Time Travel Debugging for C/C++, https://undo.io/products/udb.

[34] Jianwu Wang, Daniel Crawl, Shweta Purawat, Mai H. Nguyen, and Ilkay Altintas. 2015. Big data provenance: Challenges, state of the art and opportunities. In *2015 IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, October 29 - November 1, 2015*. IEEE Computer Society, 2509–2516. https://doi.org/10.1109/BIGDATA.2015.7364047

[35] Zuozhi Wang, Yicong Huang, Shengquan Ni, Avinash Kumar, Sadeem Alsudais, Xiaozhen Liu, Xinyuan Lin, Yunyan Ding, and Chen Li. 2024. Texera: A System for Collaborative and Interactive Data Analytics Using Workflows. *Proc. VLDB Endow.* 17, 11 (2024), 3580–3588. https://www.vldb.org/pvldb/vol17/p3580-wang.pdf

[36] 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

[37] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2022. Database Meets Artificial Intelligence: A Survey. *IEEE Trans. Knowl. Data Eng.* 34, 3 (2022), 1096–1116. https://doi.org/10.1109/TKDE.2020.2994641