Texera: A System for Collaborative and Interactive Data Analytics Using Workflows

Zuozhi Wang
UC Irvine
zuozhiw@ics.uci.edu

Yicong Huang
UC Irvine
yicongh1@ics.uci.edu

Shengquan Ni
UC Irvine
shengqun@ics.uci.edu

Avinash Kumar
UC Irvine
avinask1@ics.uci.edu

Sadeem Alsudais
UC Irvine
salsudai@ics.uci.edu

Xiaozhen Liu
UC Irvine
xiaozl3@ics.uci.edu

Xinyuan Lin
UC Irvine
xinyual3@ics.uci.edu

Yunyan Ding
UC Irvine
yunyad1@ics.uci.edu

Chen Li
UC Irvine
chenli@ics.uci.edu

ABSTRACT

Domain experts play an important role in data science, as their knowledge can unlock valuable insights from data. As they often lack technical skills required to analyze data, they need collaborations with technical experts. In these joint efforts, productive collaborations are critical not only in the phase of constructing a data science task, but more importantly, during the execution of a task. This need stems from the inherent complexity of data science, which often involves user-defined functions or machine-learning operations. Consequently, collaborators want various interactions during runtime, such as pausing/resuming the execution, inspecting an operator's state, and modifying an operator's logic. To achieve the goal, in the past few years we have been developing an open-source system called Texera to support collaborative data analytics using GUI-based workflows as cloud services. In this paper, we present a holistic view of several important design principles we followed in the design and implementation of the system. We focus on different methods of sending messages to running workers, how these methods are adopted to support various runtime interactions from users, and their trade-offs on both performance and consistency. These principles enable Texera to provide powerful user interactions during a workflow execution to facilitate efficient collaborations in data analytics.

PVLDB Reference Format:
Zuozhi Wang, Yicong Huang, Shengquan Ni, Avinash Kumar, Sadeem Alsudais, Xiaozhen Liu, Xinyuan Lin, Yunyan Ding, and Chen Li. Texera: A System for Collaborative and Interactive Data Analytics Using Workflows. PVLDB, 17(11): 3580 - 3588, 2024.
doi:10.14778/3681954.3682022

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Texera/texera.

1 INTRODUCTION

In many data science tasks, domain experts, such as public health scientists and medical researchers, play a key role because they possess valuable knowledge, such as public health policies or disease conditions. This knowledge can unlock the full potential of data-driven insights. However, they often lack the technical skills to analyze data, such as proficiency in programming languages (e.g., Python, R), familiarity with visualization techniques, and understanding of machine learning algorithms. They need collaboration with technical experts who have the necessary skills but not the domain knowledge. This need calls for systems that support collaborative data analytics by users with different backgrounds.

Collaborative editing services such as Google Docs have revolutionized how people work together. Unlike collaborative document editing, a unique aspect of collaborative data analytics is that it requires sharing not only in the editing process but more importantly, throughout the execution of an analytical job. This necessity arises from the nature that data science is a highly iterative process, and users need to go through a long trial-and-error process to refine their analysis tasks. In many data systems, analytical jobs are submitted to a backend engine and left to run until completion before any results are returned. This computing paradigm is inadequate for collaboration because users suffer from (1) lengthy delays due to having to rerun jobs after identifying errors post-execution; and (2) not being able to invite collaborators during a job execution.

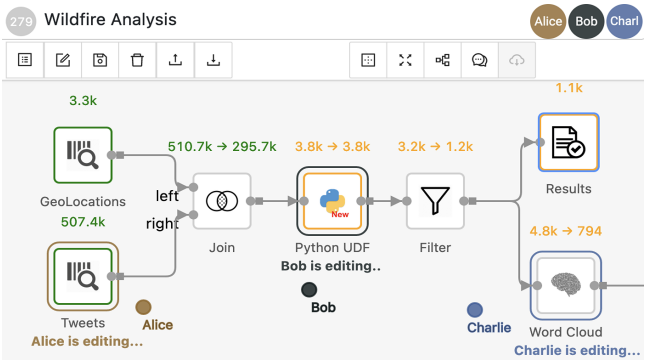


Figure 1: A Texera interface with three collaborators working on the same workflow to do social media analysis.

Therefore, it is increasingly important for data analytics systems to support collaborations and interactions throughout execution. They should allow multiple users to monitor the execution status, observe operator metrics, and view early outputs and errors. If issues are identified during the execution, users should work together to diagnose and troubleshoot the issues. The system should allow

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682022

users to perform deep investigations during the execution, such as pausing the execution, inspecting tuples, examining the state of operators, and changing the logic on the fly. This level of interactivity facilitates a more agile approach to data science, empowering teams to collaboratively make adjustments in real-time.

To achieve this goal, we have been developing Texera [25], a workflow-based data analytics system supporting collaborative data science by users from various backgrounds. As shown in Figure 1, Texera allows multiple users to collaboratively construct and execute data science projects as workflows [17], offering an experience similar to Google Docs. Texera is open-source and has been used by over 340 users across various domains, such as neuroscience and diabetes research. At the time of publishing, these users have developed more than 3,200 workflows, edited over 273,000 versions, and executed workflows over 22,000 times. It has multiple deployments, including one on a cluster of 100 quad-core nodes.

We face several challenges to support user interactions during workflow execution. First, at the workflow level, concurrent execution of multiple operators has the nature of distributed computation. We need to consider not only performance when these operators communicate with each other but also consistency when each operator takes its own actions. Second, within each operator, it needs to respond to an interaction request quickly, without introducing a significant overhead to its normal data processing.

In this paper, we present a holistic view of several important design principles we followed in the design and implementation of the system to address the challenges and achieve the goal. The paper is organized as follows. Section 2 explains user collaboration experiences in Texera. Section 3 focuses on handling interactions at the workflow level, and discusses how the coordinator dispatches interaction requests to operators. We present two methods to send control messages and analyze their trade-offs in terms of responsiveness, efficiency, and consistency when supporting three example user interactions. Section 4 discusses interaction handling at the operator level. Section 5 reports our experimental results on real-world data sets to show Texera's interactivity and scalability.

1.1 Related Work

Traditional workflow-based systems such as Alteryx [1], KNIME [5], and RapidMiner [6] do not support user interactions during execution. Some Python-based Jupyter notebooks [2, 3] and SQL notebooks [4] offer real-time collaboration features. They are mainly for programmers, not users with limited coding skills. Systems such as Dremel [20], Drill [13], and Druid [30] can efficiently process interactive OLAP queries and deliver results in seconds. Texera supports the interactivity of long-running queries, including those involving machine learning and user-defined functions (UDFs). Moreover, big data systems such as Spark [31], Hive [26], and Dask [24] are built to process vast datasets with a primary emphasis on optimizing runtime efficiency and scalability. These systems generally do not provide mechanisms for user interaction with the runtime engine.

Previously, we published how Texera's backend engine called Amber supports debugging [16], how to do run-time reconfiguration using a technique called Fries [29], and how to do line-by-line debugging in Python UDFs using a technique called Udon [14]. In this paper, we present a holistic view of the design principles in Texera to support a variety of user interactions. They were not described in the previous papers that focused on specific techniques. Furthermore, this paper analyzes different approaches to handling user interactions and gives an in-depth insight into the trade-offs regarding efficiency, semantics, and consistency.

2 TEXERA SYSTEM OVERVIEW

We first describe the experience when multiple Texera users collaboratively construct a workflow and interact with its execution, then give an overview of the system architecture.

2.1 User Experience of Collaboration

Collaborative Workflow Construction. Consider an interdisciplinary research project in analyzing the impact of wildfires on the environment using tweets [15], co-led by machine learning (ML) expert Alice and public health scientist Bob using Texera. Alice and Bob work together to create a workflow using their browsers. Bob focuses on setting up data sources and simple preprocessing steps, such as adding filters on specific locations and keywords of tweets. As an ML expert, Alice works on advanced ML operations, such as using a Python UDF to do sentiment analysis. Alice and Bob can jointly edit the same operator, such as the Python code, and they can see each other's edits in real-time. This interface facilitates a seamless integration of their complementary expertise.

Collaborative Execution, Interaction, and Debugging. Alice and Bob then proceed to execute the workflow. They monitor the execution status in real-time, such as the number of processed tuples and the average processing time per operator. If Alice sees problems during the execution, she collaborates with Bob to troubleshoot the issue. For instance, to understand the run-time status, Alice can pause the execution and examine the content of tuples, or check the value of a variable in an operator. Bob, on his end, can see the workflow is paused by Alice, as well as Alice's actions to read operator states, and the system's responses. The two users can also invite another collaborator, say, Charlie, to join the debugging process. Upon his entry into the session, Charlie can see the same information visible to Alice and Bob, including the workflow, its execution status, and the interactions by Alice and Bob.

2.2 System Architecture

Figure 2 shows Texera's architecture, consisting of frontend UI, web server, and execution engine. Next, we describe each component.

Frontend UI and Web Server. Texera offers a web-based graphical user interface (GUI) for users to construct workflows using intuitive drag-and-drop operations. To support collaborative concurrent editing by multiple users, Texera uses the JavaScript library Yjs [22] based on conflict-free replicated data types (CRDTs) [23] to resolve concurrent editing conflicts. The shared editing server relays and propagates editing changes of a user client to the interfaces of other clients. The shared execution manager handles user interaction requests during a workflow execution. In particular, it sends requests to the engine, collects responses from the engine, and broadcasts user interaction activities to other clients.

Execution engine. A workflow is executed on the engine called Amber [16] based on the actor model [7]. A workflow is a directed acyclic graph (DAG) of operators. A compiler compiles a workflow

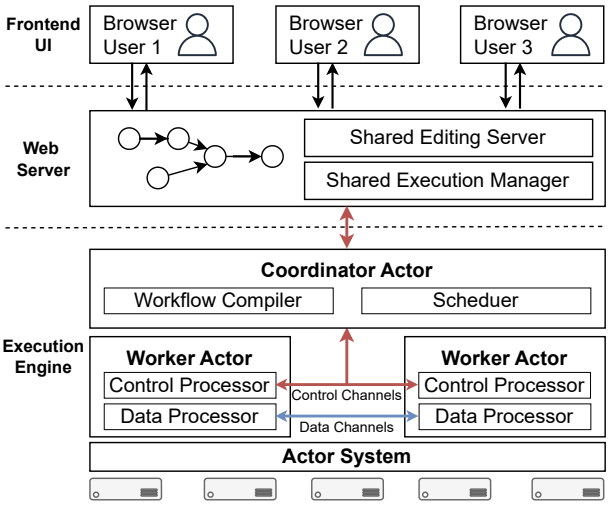


Figure 2: System architecture of Texera.

into a parallel physical execution plan. A scheduler instantiates a worker actor for each parallel instance of an operator. A coordinator actor manages the execution of worker actors. The coordinator actor has control channels with workers to exchange control messages. Workers have data channels with each other to send data tuples. Each worker actor has a control processor to handle control messages and a data processor to execute its operator logic, which will be explained in Section 4. Amber is a push-based engine and executes workflows in a pipelined fashion. Pipelining allows the concurrent computation of multiple operators and lowers the end-to-end latency, enabling users to observe the results sooner.

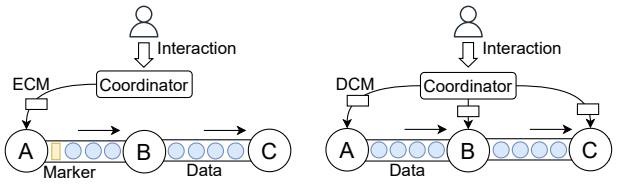
3 HANDLING USER INTERACTIONS AT THE WORKFLOW LEVEL

In this section, we discuss how Texera handles interactions at workflow level. We present two methods of sending control messages and analyze how they are used to support various user interactions.

3.1 Two Methods of Sending Control Messages

User interactions in Texera are received by the workflow coordinator and sent as control messages to workers. A *control message*, as opposed to a data message carrying data tuples, is a message that includes instructions for workers to take "control-related" actions, such as pausing its execution, reading its internal state, and modifying its logic. We consider two methods to send control messages.

Method 1: embedding control messages in data streams from sources. This method embeds a control message as a special message, also referred to as a "marker," and propagates it in the data streams [9]. We call such a control message an *embedded control message*, or "ECM" for short. After receiving an interaction from the user, the coordinator sends a control message to each source worker, which injects a special ECM marker and sends the generated ECM to their downstream worker(s) following the order of its output data messages. When a worker receives an ECM, it performs marker alignment by waiting for all its input edges to receive this ECM before processing it. After that, the worker sends the ECM to its downstream worker(s). Figure 3a shows how a control message is embedded into the data stream from a source worker A.



(a) Method 1. Using embedded control messages (ECMs). **(b) Method 2. Using direct control messages (DCMs).**

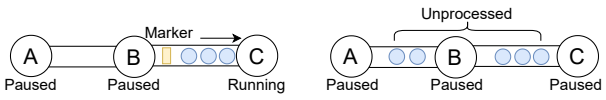
Figure 3: Two methods of sending control messages from the coordinator to workers A, B, and C to handle an interaction.

Method 2: sending direct control messages. Using this method, the coordinator sends direct control messages to target workers using dedicated channels. A worker does not send additional control messages to its downstream. Such a control message is called a *direct control message* (DCM). Upon receiving a DCM, a worker gives a higher priority to process it, and sends a response back to the coordinator. We will discuss the prioritization mechanism in Section 4. Figure 3b shows how DMCs are used to send control messages. Next, we discuss how to use these two methods to support three example runtime interactions and analyze their pros and cons.

3.2 Pausing the Workflow Execution

Texera supports pausing workflows on the fly and allows for additional interactions while paused. For instance, after Alice monitors the workflow execution for a while, she notices the execution has become very slow. She pauses the execution to ensure the workflow is in a stable state, then asks Bob to perform further investigations.

Using ECMs. The coordinator sends a Pause control message to each source worker. Upon receiving this message, a source worker pauses emitting data. After that, it emits the Pause ECM to its downstream workers. When a downstream worker finishes processing all its pending data tuples and receives a Pause message from all its upstream workers, its computation is paused because the upstream workers no longer send any data tuples. Then it propagates the Pause message using the ECM markers to its downstream workers. Figure 4a shows an execution state where workers A and B are paused by the marker, but worker C is still processing data.



(a) Using ECM. **(b) Using DCM.**

Figure 4: Using ECM, workers A and B have paused when worker C is still waiting for the marker. Using DCM, all workers have paused, leaving in-flight data unprocessed.

Using DCMs. The coordinator sends a Pause control message directly to all the workers. Upon receiving this message, each worker stops processing incoming data messages but can continue processing control messages from the control channel. As the message-delivery time can vary, there can be a situation where, for a specific

data channel, the receiver-side worker is paused, but the sender-side worker is still processing data and outputting data messages. We refer to *in-flight messages* as those sent by the sender but not yet processed by the receiver. This results in an accumulation of more in-flight data messages in the receiver's input buffer. Figure 4b shows the state after all workers are paused, and there are in-flight messages left unprocessed between workers.

Comparison of the two methods. In the ECM-based method, there are no in-flight messages in the channels between workers in a paused state. This is an advantage over the DCM-based method, which may have increased memory usage due to unprocessed in-flight messages. However, the ECM-based method has a much higher pause latency as the system needs to fully process all the in-flight data before transitioning to the paused state. This latency could be even longer if the workflow contains expensive workers or has a large number of in-flight tuples. Texera adopts the DCM-based method because it offers a low pause latency, which is mainly decided by network speeds. Several deployments of Texera show that the latency of this method tends to be within a second.

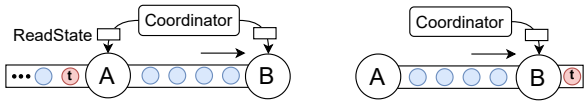
To the best of our knowledge, most data-processing systems do not support pausing a workflow on the fly nor support interactions in the paused state. Many systems support checkpointing of a workflow [10, 21, 27], which can be considered analogous to stopping the workflow and then restarting it. However, checkpointing is usually a slow and resource-intensive process. In contrast, Texera's pausing mechanism is lightweight and fast, offering an advantage in scenarios requiring frequent interactions with the workflow.

3.3 Reading Workers' Internal States

Texera supports reading the internal states of workers during an execution, providing users with valuable insights into the process. For example, when Bob investigates why the workflow has slowed down, he might check each operator's state, processing speed, and the content of in-flight tuples. When the workflow is paused, Bob can check these states. He can also check them while the workflow is running, thus having an interactive experience.

Using DCMs. The coordinator directly sends a ReadState control message to every worker, which responds by sending its current internal state back to the coordinator.

Inconsistent global states captured by DCMs. This method could lead to an "inconsistent" global state, as defined by Chandy and Lamport [11]. That is, an inconsistent global state occurs when a process's state reflects a message receipt, but the state of the corresponding sender does not reflect sending that message. For



(a) Worker A reports its state before processing tuple t. **(b) Worker B reports its state after processing tuple t.**

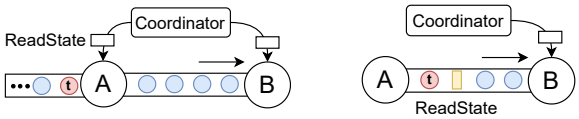
Figure 5: Inconsistent state between workers A and B.

instance, consider a scenario where the ReadState DCM is sent to two workers, A and B, with the following sequence of events: (1) Worker A receives the DCM and reads its state, as shown in Figure 5a; (2) Worker A processes a tuple t, sending it to worker B; (3) Worker B processes tuple t and updates its state; (4) Due to a network delay, worker B receives the ReadState DCM from the coordinator and reads its state, as shown in Figure 5b. This sequence of events results in a discrepancy where worker A's state does not reflect the processing and sending of tuple t, whereas worker B's state shows the receipt of t, leading to an inconsistent global state.

Combining ECMs and DCMs to read workers' states. Texera adopts a hybrid strategy, leveraging the strengths of both methods by having the coordinator utilize both mechanisms within the workflow for interactions. In particular, for an interaction, the coordinator generates a control message with a unique ID and then dispatches it as DCMs to all workers. Upon receiving a control message, a worker processes the message and returns its internal state. After that, the worker sends the control message as ECMs to all its downstream workers. In this case, a worker may receive both the DCM from the coordinator and an ECM from each of its upstream workers, with all messages sharing the same ID. The worker will only process the first instance of such a message, and ignore the subsequent messages with the same control message ID.

This hybrid method ensures a consistent global state. Recall that in the DCM approach, inconsistency arises when the DCM sent to the worker B is delayed. For example, in Figure 6, tuple t is processed by worker A after worker A receives the ReadState DCM. Worker B hasn't received the ReadState DCM yet due to possible network delays. If B were to process t before reading its state, inconsistency would occur. To prevent this, the hybrid mechanism ensures worker A immediately sends an ECM to B after receiving the ReadState DCM, before processing and emitting the next tuple t. Because ECMs follow the order of data, this ECM will always reach B faster and ensure B's state is read before it processes the tuple t.

By propagating ECMs between workers, a downstream worker always receives an ECM before processing any tuples not reflected in its upstream worker's state. By sending DCMs, this approach ensures that reading the state is also fast in most cases because DCMs typically reach workers faster due to the direct channel. This approach offers the benefits of both consistency and low latency, but at the cost of transmitting additional control messages.



(a) Worker A reports its state before processing tuple t. **(b) A** ReadState **message is embedded before tuple** t.

Figure 6: Combining ECMs and DCMs to read states.

Recall that we pause workflows with only DCMs. After the workflow is fully paused, DCMs can also ensure reading a consistent state of operators. As no data is sent and processed in the paused state, a downstream worker cannot process any data not yet reflected in its upstream workers' states when paused.

Many systems support reading the state of operators, especially in streaming settings [8, 9, 19]. These systems support taking a consistent snapshot of the workflow state and allow users to read the snapshot, but this process is slow. Another method [28] supports querying the live state by maintaining each operator's state in an external state backend, which can be queried directly. Such direct

reading is similar to DCMs since it does not follow the order of data as in ECMs. However, this method faces the same issue of potentially reading an inconsistent state as DCMs do. Texera ensures reading a consistent state using the hybrid approach.

3.4 Modifying Workers' Logic

Texera supports modifying the logic of workers during an execution, also referred to as "reconfigurations," allowing users to correct loopholes or integrate new logic for unexpected data formats. We discuss two real-world use cases: one where the user performs a reconfiguration on the fly, and another where the workflow execution is paused due to an error, prompting the user to modify the logic to fix the issue. We examine scenarios where multiple workers are modified, focusing on the possibility of processing a tuple with a mix of old and new logic. Additionally, we discuss the associated consistency levels, their semantics, and implementation.

3.4.1 Reconfiguration without pausing. Consider a scenario where Alice and Bob have a data science pipeline. An upstream encoder worker X encodes a tweet for a machine learning model, followed by a corresponding downstream decoder worker Y to decode it back to the original format. After investigating the cause of the slow execution, Bob decides to modify the logic of both workers to use a faster implementation. He aims to accomplish this without pausing the workflow to minimize disruption.

To perform a reconfiguration, the coordinator sends a ModifyLogic message to each target worker. The ModifyLogic message contains the new tuple-processing function to replace the old logic, and optionally a state-transfer function to convert the old internal state of the worker to the new state required by the new logic. After receiving the message, a worker applies the state-transfer function and updates its logic. Depending on the arrival order of the ModifyLogic messages, there are four possible scenarios (marked as 1 - 4) for an input tuple processed by both workers X and Y, as shown in Table 1. This can be generalized to any reconfiguration involving two workers, with X being the upstream worker of Y. We consider different consistency levels based on which scenarios are allowed.

Strict consistency: At this level, an input tuple can be processed by operators X and Y using either both old versions or both new versions. This level is the strictest and the least error-prone. It is equivalent to pausing the workflow, stopping the ingestion of new tuples from the source, waiting for all the operators to finish processing all in-flight tuples, and then performing the reconfiguration. To support this level, the reconfiguration request can be carried out using ECMs. A ModifyLogic message is sent to each source operator and then propagated downstream following the order of data. The ECM acts as a barrier that separates tuples processed by the old logic versus the new logic: tuples arriving at the source before the ModifyLogic message will use the old logic of the entire workflow, and tuples arriving after will use the new logic.

Backward-compatible consistency: This level additionally allows an input tuple to be processed by the upstream operator X using its old logic but processed by the downstream operator Y using its new logic. This level is acceptable if Y's logic is "backward-compatible," i.e., it can gracefully handle tuples processed by the old logic of X. It is equivalent to first pausing the workflow, allowing some in-flight tuples between operators when paused, and then performing the reconfiguration. To implement this level, we can use the same hybrid mechanism that combines ECMs and DCMs in Section 3.3, with the ReadState message replaced by the ModifyLogic message. This mechanism applies control messages on top of a consistent snapshot of the workflow, which might contain in-flight tuples between operators. For instance, the in-flight tuples between X and Y are processed by X using the old logic and are processed by Y using the new logic.

Table 1: Consistency levels and allowed scenarios for modifying the logic of worker X and its downstream worker Y.

Consistency Levels	Scenario 1		Scenario 2		Scenario 3		Scenario 4	
	X	Y	X	Y	X	Y	X	Y
	new	new	old	new	new	old	old	old
Strict Consistency	✓						✓	
Backward-Compatible	✓		✓				✓	
Forward-Compatible	✓				✓		✓	
No Consistency	✓		✓		✓		✓	

Forward-Compatible Consistency: This level allows a tuple to be processed by the new version of the upstream worker X and the old version of the downstream worker Y. This requires Y's code to be "forward-compatible," meaning that its old version can handle a tuple processed by the new version of X. Texera does not support this level, as we believe it is rare in real-world scenarios to permit only forward-compatibility without backward-compatibility.

No Consistency: This level has no restrictions and allows all scenarios to occur. Texera supports this level by sending ModifyLogic messages using DCMs. While users must carefully ensure their logic updates are correct, this approach is the fastest and has the lowest overhead due to the use of only DCMs.

3.4.2 Reconfiguration when encountering an error. Consider another scenario where the workflow is paused due to an error. Suppose a problematic tuple with an unexpected format arrives. The tuple is first processed by encoder X. When decoder Y processes it, Y raises an exception, e.g., due to an integrity-check failure. The coordinator then pauses all workers, allowing Bob to inspect the problematic tuple. Bob then wants to update the logic of both workers to properly re-process this tuple and any subsequent tuples.

In this scenario, using the strict consistency level is not ideal, because the workflow must be restarted. The problematic tuple cannot be handled using the old versions of both X and Y, as Y would raise an exception. To process the problematic tuple using the new version of X, the workflow must either roll back to the last checkpoint before this tuple was processed by X or be completely rerun with the new logic. On the other hand, backward-compatible consistency is preferred as the user can provide a backward-compatible implementation of decoder Y, which can properly handle the problematic tuple processed by the old version of X. Because this level avoids restarting the workflow, it can significantly improve the interactivity during the workflow execution and allow users to seamlessly fix the bug in an iterative process without any wait.

Supporting reconfiguration in dataflow systems has recently gained popularity. One method is stopping and restarting the workflow with new logic [12], but it is disruptive and hinders interactivity. Our prior work [29] studied the strict consistency level, provided a formal definition based on conflict serializability, and proposed an optimized algorithm that uses ECMs on a sub-region of

the graph to enhance performance. This paper extends the previous study beyond strict consistency to explore other consistency levels, which are also useful in the scenarios described above.

4 HANDLING USER INTERACTIONS AT THE WORKER LEVEL

In this section, we discuss how a worker handles a control message after receiving it. For ECMs, their processing naturally follows the sequence of the data. Prioritizing DCMs is more challenging, as the worker is actively performing its computation when a DCM arrives. We discuss two approaches to interrupting a worker's execution: forcibly interrupting its execution, and letting the worker voluntarily check for DCM messages. We use pausing and reading worker states as examples to discuss these methods and their trade-offs, which can be generalized to other interactions.

4.1 Forcibly Interrupting the Worker Execution

One approach is to use thread-level interruptions, such as Java's Thread.suspend() function. Each worker employs two threads: a data-processing (DP) thread responsible for running the computation, and a control-processing (CP) thread, which listens to user requests. As shown in Figure 7, upon receiving a message, the CP thread invokes the Thread.suspend() function to halt the DP thread's execution. This approach provides a quick and efficient way to pause and resume a worker's execution without terminating it. It also allows a user to read the internal state of a worker while it is paused, as the request can be served by the CP thread.

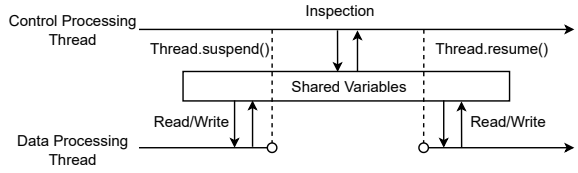


Figure 7: Using thread-level interruptions to pause a worker.

This approach has a significant drawback: users cannot decide the precise moment to pause the execution. The process could stop at any random instruction, possibly during a series of function calls in third-party libraries that are not of interest to the user. Thus this approach is not suitable for the Texera system.

4.2 Our Solution: Voluntarily Checking Interruptions at Pre-determined Points

Texera employs a design that avoids forcibly interrupting the execution of a worker. A key insight is that data-processing tasks can be broken down into discrete steps (e.g., processing a single tuple can be one discrete step), and the execution point after a step is more meaningful for users' interactions. Our approach allows workers to voluntarily check for interruptions at pre-determined stopping points in their execution. We start by discussing stopping points that occur between the processing of two tuples, and then extend to having stopping points of even finer granularity.

Checking interruptions between tuples. It is natural to conduct voluntary interruptions between the processing of two tuples. Our design for a worker incorporates two main components: the control processor and the data processor, which operate within the same thread. The two components take turns to execute in a loop: after the data processor processes one tuple, the control processor gets an opportunity to execute. As shown in Figure 8, the data processor first takes the turn to process one tuple. Then, the control processor voluntarily checks if there is a Pause control message. If so, the control processor pauses the data processor by removing it from the execution loop. When the user sends a Resume control message, the control processor resumes the data processor by adding it back to the execution loop.

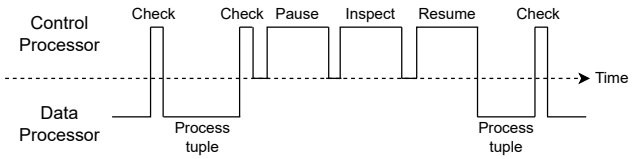


Figure 8: The control processor voluntarily checks for interruptions between the processing of two tuples.

One challenge in this method involves dealing with workers with different execution speeds. While the processing of a single tuple is typically fast for relational operations, some customized logic such as UDFs can be computationally expensive. For example, consider a sentiment analysis worker with an expensive machine learning model to classify a tuple. If the worker can only pause between processing two tuples, the user may have to wait a long time before the system pauses.

```
interface Operator {
    def processTuple(t Tuple, port int): Iterator[Tuple]
    def onFinish(port int): Iterator[Tuple]
}
```

Figure 9: Operator interface in Texera.

Checking interruptions with a finer granularity. We want to ensure a low latency in handling control messages, even for slower operators. A key observation is that a worker's processing of a tuple can be further divided into multiple mini-steps, with each mini-step performing a portion of the worker's computation. Texera provides the following operator interface for processing a single tuple, which returns an iterator of multiple mini-steps.

```
class SentimentAnalyzer extends Iterator {
  var next_step: String = "tokenize";
  def has_next(): Boolean = {return next_step != "end";}
  def get_next(tuple: Tuple): Double = {
    if (next_step == "tokenize") {
      this.tokens = tokenize(tuple);
      next_step = "tag"; return null;
    } else if (next_step == "tag") {
      this.tagged = tag(this.tokens);
      next_step = "infer"; return null;
    } else if (next_step == "infer") {
      this.sentiment = infer(this.tagged);
      next_step = "end"; return this.sentiment;
}}}
```

Figure 10: An iterator-based sentiment analysis operator that decomposes the computation into three mini-steps.

Consider the following sentiment analysis operator implementation: invoking the process_tuple function generates an iterator that implements two functions, has_next and get_next, as shown

in Figure 10. This iterator comprises three sequential mini-steps: tokenization, tagging, and inference. It runs as a state machine using the next_step variable tracking the current mini-step. To process a single tuple, the engine first acquires this iterator from the worker, and then consumes the iterator to go through each mini-step of the computation. The engine performs voluntary checks for control messages between each mini-step iteration.

Despite such fine-granularity checks being very fast, having too many of them can still lead to a noticeable overhead. Experiments in Section 5.1 show that frequent checks on a simple filter operator can have up to 25% overhead. To solve this problem, developers can leverage their knowledge of worker semantics and execution speed to define mini-steps that ensure responsiveness and provide meaningful stopping points. The system can also automatically adjust frequency, reducing it for fast operators such as filters.

Another consideration is the burden on developers to manually convert programs into state machines. Fortunately, some languages such as Python, R, JavaScript, and C# can alleviate this complexity with the `yield` keyword, which transforms a regular function into an iterator. Figure 11 shows the same logic in Figure 10 (14 lines) implemented in Python with only 6 lines. The `yield` keyword transforms the `sentiment_analyzer` function into an iterator. Unlike a regular function that runs to completion, this iterator suspends execution each time `yield` is encountered and resumes from there when `get_next` is called again. The execution states, e.g., local variables and the current line, are managed by the language runtime.

```
def sentiment_analyzer(tuple: Tuple) -> Iterator:
    tokens = tokenize(tuple)
    yield   # returns null, perform voluntary check
    tags = tag(tokens)
    yield   # returns null, perform voluntary check
    yield infer(tags) # return final result
```

Figure 11: Sentiment analyzer with Python's `yield` keyword.

5 EXPERIMENTS

Datasets. We used four datasets: (1) a geolocation dataset, with each row containing a geolocation ID and its longitude and latitude; (2) a tweet dataset consisting of 700K tweets (100MB), each containing the ID, text, and geolocation ID of a tweet; (3) a review dataset with two tables: one with 67,000 user reviews, and another with 4,500 cleaned reviews; and (4) a stock dataset containing daily closing prices of stocks from April to July 2017, and a table with synthetic trading history for 68 traders.

Workflows. We used three workflows. Workflow W_1 joined the tweet dataset with the geolocation dataset, filtered the results using keywords, calculated the sentiment of each tweet, and calculated the average sentiment. Workflow W_2 used the review dataset, removed nulls and duplicates in the uncleaned user reviews, then added the result to the cleaned reviews. Workflow W_3 used the stock dataset to compute the total return by aggregating the return for each trade after joining the trading history with the daily closing prices.

Computing environment. The experiments were conducted on a cluster of 10 virtual machines (VMs) hosted on the Google Cloud Platform (GCP). Each VM was of type e2-highmem-4, had a 100GB SSD persistent disk, and ran Ubuntu 20.04. All source and sink operators had one worker and were placed in the same VM. All other operators had two workers per VM.

5.1 Methods to Pause a Worker

We evaluated Texera's pause latency for a worker by comparing forced interruption and voluntary-check methods, and assessed the overhead of voluntary checks at different frequencies.

Comparing voluntary-check and forced interruption. We evaluated the latency of interrupting a single worker by comparing the forced-interruption and the voluntary-check methods described in Section 4 with workflow W_1. We used 1 VM and every operator had a single worker. The latency was measured from the moment when the worker received a Pause request to when the worker's computation was paused. Figure 12 shows the results. With thread-level forced interruption, all workers were paused within 1 ms. The computation-heavy workers required more time to pause. For example, the Sentiment Analysis worker was the slowest at 0.4 ms.

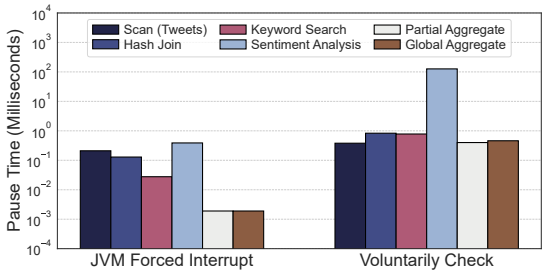


Figure 12: Comparison of interruption latency between the thread interruption and the voluntary check method.

For Texera's voluntary-check method, the pause latency depended on the execution duration of each mini-step of the operator logic. The Sentiment Analysis implementation was divided into four mini-steps, resulting in an average pause latency of roughly 126 ms. Though longer than the forced-interruption method, the voluntary-check method still delivered a responsive user experience. In summary, the thread-level forced interruption method had very low pausing latency but paused at random, potentially meaningless points. Texera uses the voluntary check method, with predefined, user-meaningful stopping points. Despite a slightly higher latency, this method achieved a latency of less than 200 ms.

Effect of voluntary-check frequency. We evaluated the overhead of the voluntary-check method using a filter worker comparing a field in a tuple to an integer, and a sentiment analysis worker implemented with Stanford NLP [18]. We varied the check frequency by adjusting the number of tuples between checks from 1, 10, 100, 1,000, to 10,000, with a baseline of no checks. Figure 13 shows the results. The filter worker had up to 25% overhead for checking every tuple, and the overhead decreased as check frequency reduced. The sentiment analysis worker had no noticeable overhead from fine-grained checks due to its much slower processing speed (100 ms per tuple vs. less than 0.7 ms for the filter). For example, in one minute, the filter operator processes around 100K tuples; checking every tuple results in 100K checks. As a result, the impact of the constant-time voluntary checks is more pronounced. For filter, checking every 10 tuples results in around 10,000 checks, significantly lowering the overhead. The sentiment analysis operator processes around 600 tuples, resulting in about 600 checks even at the highest check frequency, thus its overhead is always low.

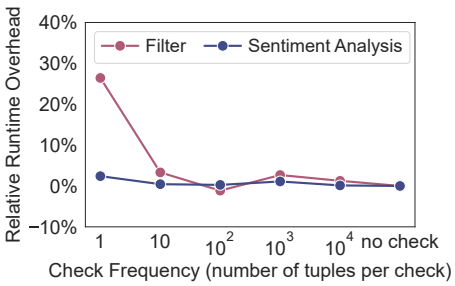


Figure 13: Overhead for different check frequencies in W_1.

5.2 Methods to Pause a Workflow

We compared the latency to pause workflows W_1, W_2, and W_3 using the two methods described in Section 3.1. During execution, a Pause request was sent every 10 seconds. We recorded the time from sending each request till the workflow was completely paused as the pause latency. We varied the cluster size from 1 VM to 10 VMs. Figure 14 shows the results. As the number of VMs increased, the latency increased for both methods. For workflow W_1, the DCM method had a low latency, peaking at around 1.2 seconds with 10 VMs. The ECM method had a high latency (from 5 seconds with 1 VM to 13.7 seconds with 10 VMs). For workflow W_2, the ECM method initially had a latency comparable to the DCM method because all operators in this workflow were inexpensive. Thus the ECM marker quickly passed to downstream workers. However, the ECM method had a high latency with more VMs. For workflow W_3, the ECM method consistently exhibited a similar latency because the join operation produced a large number of output tuples, which blocked the processing of the ECM marker. Overall the DCM method achieved a much lower latency because it did not require the control messages to wait for data processing.

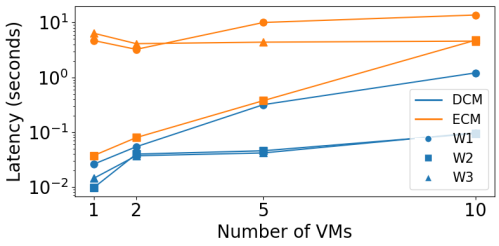


Figure 14: Comparing two methods to support Pause interactions on workflows W_1, W_2, and W_3.

We compared the memory overhead of pausing with the DCM approach, which left in-flight tuples in channels between operators, versus the ECM approach, where there were no in-flight messages. Table 2 shows the total size of in-flight messages after each workflow was paused. The size of in-flight messages for the DCM method increased with the cluster size. This was because a shuffling operation between two operators created a full many-to-many channel, e.g., with 10 machines, there were 100 channels to shuffle the data. Workflow W_1 has two operators that require shuffling, join, and aggregate, resulting in a higher size of in-flight tuples when paused. Workflows W_2 and W_3 had fewer in-flight tuples but followed a similar trend.

Table 2: In-flight message sizes in megabytes for DCM and ECM across different workflows and number of VMs

	1 VM	2 VMs	5 VMs	10 VMs
W1 w/ DCM	0.07	0.1	1519	17183
W1 w/ ECM	0	0	0	0
W2 w/ DCM	1.48	4	19	407
W2 w/ ECM	0	0	0	0
W3 w/ DCM	1.01	1.32	8.16	92
W3 w/ ECM	0	0	0	0

5.3 Methods to Read and Modify Operator States

In this experiment, we evaluated handling user interactions using the DCM approach and the hybrid approach. Both reading state and modifying logic could be implemented using the same mechanism, differentiated only by the control messages (`ReadState` or `ModifyLogic`). We sent dummy control messages that performed no action to simulate the handling of `ReadState` and `ModifyLogic` messages. The dummy messages were sent every 10 seconds, and we measured the average latency to process them. We showed the results in Figure 15. For W_2 and W_3, the latency of the DCM approach (around 100 ms) was lower compared to the hybrid approach (around 1 second). This was expected because the DCM approach transmits far fewer control messages. However, it was important to note that the DCM approach offered no consistency in reading state or modifying logic, whereas the hybrid approach guarantees a consistent state and provides backward-consistency when modifying logic. For W_1, the latency of both the hybrid and DCM approaches were similar. This was due to the higher number of shuffling operations in W_1, where the additional latency of control messages was dominated by data-processing overhead.

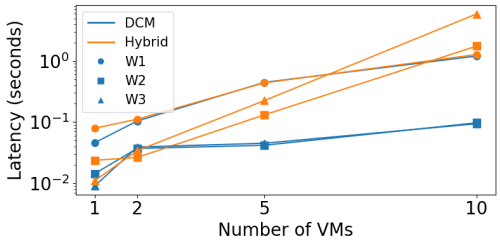


Figure 15: Latency of a user interaction to read and modify states of operators in workflows W_1, W_2, and W_3.

6 CONCLUSIONS

In this paper, we gave a holistic view of the design principles in the open-source Texera system to support collaborations and a variety of interactions throughout workflow execution. We presented different methods of sending control messages to running workers, and trade-offs of using these methods to support three common types of interactions, namely pausing an execution, reading an operator's state, and modifying an operator's logic. We reported experimental results on real-world datasets to evaluate these techniques.

ACKNOWLEDGEMENTS

This work was supported by NSF awards IIS-1745673 and IIS-2107150.

REFERENCES

[1] 2024. Data Science and Analytics Automation Platform | Alteryx — alteryx.com. https://www.alteryx.com/.

[2] 2024. Deepnote: Analytics and data science notebook for teams. — deepnote.com. https://deepnote.com/.

[3] 2024. Google Colab — research.google.com. https://research.google.com/colaboratory/faq.html.

[4] 2024. Introduction to Databricks notebooks | Databricks on AWS — docs.databricks.com. https://docs.databricks.com/notebooks/index.html.

[5] 2024. Open for Innovation | KNIME — knime.com. https://www.knime.com/.

[6] 2024. RapidMiner | Amplify the Impact of Your People, Expertise and Data — rapidminer.com. https://rapidminer.com/.

[7] Gul A. Agha. 1990. *ACTORS - a model of concurrent computation in distributed systems.* MIT Press.

[8] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[9] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[10] 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

[11] 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

[12] FlinkSavepoint [n.d.]. Savepoints in Apache Flink, https://ci.apache.org/projects/flink/flink-docs-master/docs/ops/state/savepoints/.

[13] Michael Hausenblas and Jacques Nadeau. 2013. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big Data* 1, 2 (2013), 100–104. https://doi.org/10.1089/BIG.2013.0011

[14] 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

[15] Jessie W. Y. Ko, Shengquan Ni, ALEXANDER TAYLOR, Xiusi Chen, Yicong Huang, Kumar Avinash, Sadeem Alsudais, Zuozhi Wang, Xiaozhen Liu, Wei WANG, Chen Li, and Suellen Hopfer. 2024. How the experience of California wildfires shape Twitter climate change framings. *Climatic Change* 177, 1 (jan 2024). https://doi.org/10.1007/s10584-023-03668-0

[16] 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

[17] Xiaozhen Liu, Zuozhi Wang, Shengquan Ni, Sadeem Alsudais, Yicong Huang, Avinash Kumar, and Chen Li. 2022. Demonstration of Collaborative and Interactive Workflow-Based Data Analytics in Texera. *Proc. VLDB Endow.* 15, 12 (2022), 3738–3741. https://www.vldb.org/pvldb/vol15/p3738-liu.pdf

[18] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations.* The Association for Computer Linguistics, 55–60. https://doi.org/10.3115/v1/p14-5010

[19] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Çetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. 2015. S-Store: Streaming Meets Transaction Processing. *Proc. VLDB Endow.* 8, 13 (2015), 2134–2145. https://doi.org/10.14778/2831360.2831367

[20] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[21] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* 439–455.

[22] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2015. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In *Engineering the Web in the Big Data Era - 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings (Lecture Notes in Computer Science)*, Philipp Cimiano, Flavius Frasincar, Geert-Jan Houben, and Daniel Schwabe (Eds.), Vol. 9114. Springer, 675–678. https://doi.org/10.1007/978-3-319-19890-3_55

[23] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. 2019. Conflict-Free Replicated Data Types CRDTs. In *Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Y. Zomaya (Eds.). Springer. https://doi.org/10.1007/978-3-319-63962-8_185-1

[24] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference 2015 (SciPy 2015), Austin, Texas, July 6 - 12, 2015*, Kathryn Huff and James Bergstra (Eds.). scipy.org, 126–132. https://doi.org/10.25080/Majora-7b98e3ed-013

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

[26] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras (Eds.). IEEE Computer Society, 996–1005. https://doi.org/10.1109/ICDE.2010.5447738

[27] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. 2014. Storm@twitter. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 147–156. https://doi.org/10.1145/2588555.2595641

[28] Jim Verheijde, Vassilios Karakoidas, Marios Fragkoulis, and Asterios Katsifodimos. 2022. S-QUERY: Opening the Black Box of Internal Stream Processor State. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022.* IEEE, 1314–1327. https://doi.org/10.1109/ICDE53745.2022.00103

[29] Zuozhi Wang, Shengquan Ni, Avinash Kumar, and Chen Li. 2022. Fries: Fast and Consistent Runtime Reconfiguration in Dataflow Systems with Transactional Guarantees. *Proc. VLDB Endow.* 16, 2 (2022), 256–268. https://doi.org/10.14778/3565816.3565827

[30] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: a real-time analytical data store. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 157–168. https://doi.org/10.1145/2588555.2595631

[31] 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