Amber: A Debuggable Dataflow System Based on the Actor Model

Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li
Department of Computer Science, UC Irvine, CA 92697, USA
{avinask1, zuozhiw, shengqun, chenli}@ics.uci.edu

ABSTRACT

A long-running analytic task on big data often leaves a developer in the dark without providing valuable feedback about the status of the execution. In addition, a failed job that needs to restart from scratch can waste earlier computing resources. An effective method to address these issues is to allow the developer to debug the task *during* its execution, which is unfortunately not supported by existing big data solutions. In this paper we develop a system called Amber that supports responsive debugging during the execution of a workflow task. After starting the execution, the developer can pause the job at will, investigate the states of the cluster, modify the job, and resume the computation. She can also set conditional breakpoints to pause the execution when certain conditions are satisfied. In this way, the developer can gain a much better understanding of the run-time behavior of the execution and more easily identify issues in the job or data. Amber is based on the actor model, a distributed computing paradigm that provides concurrent units of computation using actors. We give a full specification of Amber, and implement it on top of the Orleans system. Our experiments show its high performance and usability of debugging on computing clusters.

PVLDB Reference Format:
Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. Amber: A Debuggable Dataflow System Based on the Actor Model. *PVLDB*, 13(5): 740-753, 2020.
DOI: https://doi.org/10.14778/3377369.3377381

1. INTRODUCTION

As information volumes in many applications continuously grow, analytics of large amounts of data is becoming increasingly important. Many big data engines have been developed to support scalable analytics using computing clusters. In these systems, a main challenge faced by developers when running an analytic task on a large data set is its long running time, which can take hours, days, or even weeks. Such a long-running task often leaves the developer in the dark without providing valuable feedback about the status of the execution [13]. What is worse is that the job can fail due to various reasons, such as software bugs, unexpected data, or hardware issues. In the case of failure, earlier computing resources and time are wasted, and a new job needs to be submitted from scratch.

Analysts have resorted to different techniques to identify errors in job execution. One could first run a job on a small data set, with the hope of reproducing the failure, finding and solving the problems before running it on the entire data set. Unfortunately, many run-time failures occur only on a big data set. For instance, a software bug is triggered only by some rare, outlier data instances, which may not appear in a small data set [19, 15]. As another example, there can be an out-of-memory (OOM) exception that happens only when the data volume is large.

Another method is to instrument the software to generate log records to do post-execution analysis. This approach has several limitations. First, the developer has to add statements at many places in order to find bugs. These statements can produce an inordinate amount of log records to be analyzed offline, and most of them are irrelevant. Second, these log records may not reveal all the information about the run-time behavior of the job, making it hard to identify the errors. This situation is similar to the scenario of debugging a C program. Instead of using `printf()` to produce a lot of output messages and do post-execution analysis, many developers prefer to use a debugger such as `gdb` to investigate the run-time behavior of the program *during* its execution.

The aforementioned shortcomings of the debugging techniques have led data analysts to seek more powerful monitoring and debugging capabilities [30, 8, 13]. There are several recent efforts to provide debugging capabilities to big data engines [15, 16]. As an example, BigDebug [15] used a concept of *simulated breakpoint* during the execution of an Apache Spark job. Once the execution arrives at the breakpoint, the user can inspect the program state. More details about these approaches and their limitations are discussed in Section 1.1. A fundamental reason of their limitations is that they are developed on engines such as Spark that are not natively designed to support debugging capabilities, which limit their performance and usability.

In this paper, we consider the following question: can we develop a scalable data-processing engine that supports responsive debugging? We answer the question by developing a parallel data-processing system called Amber, which stands for "actor-model-based debugger." A user of the sys-

Proceedings of the VLDB Endowment, Vol. 13, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3377369.3377381

tem can interact with an analytic job during its execution. For instance, she can pause the execution, investigate the states of operators in the job, and check statistics such as the number of processed records and average time to process each record in an operator. Even if the execution is paused, she can still interact with the operators in the job. The user can modify the job, e.g., by changing the threshold in a selection predicate, a regular expression in an entity extractor operator, or some parameters in a machine learning (ML) operator. The user can also set conditional breakpoints, so that the execution can be paused automatically when a condition is satisfied. Examples of conditions are incorrect input formats, occurrences of exceptions etc. In this way, the user can skip many irrelevant iterations. After doing some investigation, she can resume the execution to process the remaining data. To our best knowledge, Amber is the first system with these debugging capabilities.

Amber is based on the *actor model* [20, 1], a distributed computing paradigm that provides concurrent units of computation called *actors*. The message-passing mechanism between actors in the actor model makes it easy to support both data messages and debugging requests, and allows low-latency control-message processing. Also after the execution of a workflow is paused, the actor-based operators can still receive messages and respond to user requests. More details about the actor model and the motivation behind using it for Amber are described in Section 2.2.

The actor model has been around for decades and there are data-processing frameworks built on top of it [29, 23]. A natural question is "why do we develop Amber now?". The answer is twofold. First, as data is getting increasingly bigger, the need for a system that supports responsive debugging during big data processing is getting more important. Second, there are more mature and widely adopted actor model implementations on clusters recently, making it easy to develop our system without reinventing the wheel.

There are several challenges in developing Amber using the actor model. First, every actor has a single mailbox, which is a FIFO queue storing both data messages and control messages. (The actor model does not support priority messages natively.) Large-scale data processing implies that data messages sent to an actor can be significantly more than its incoming control messages. Thus, the mailbox can already have many data messages when a control message arrives. Responsive debugging requires that control messages be processed quickly, but the control message can only be processed after those data messages ahead of it. Second, a data message can take an arbitrarily long time to process (e.g., in an expensive ML operator). Real-time debugging necessitates that user requests should be taken care of in the middle of processing a data message instead of waiting for the entire message to be processed, which could take a long time depending on the complexity of the operator.

In this paper we tackle these challenges and make the following contributions. In Section 2 we discuss important features related to debugging the execution of a data workflow, and analyze the requirements of an engine to support these features. In Section 3 we present the overall architecture of Amber and study how to construct an actor workflow for an operator workflow, how to allocate resources to actors, and how to transfer data between actors. In Section 4 we describe the lifecycle of executing a workflow, discuss how control messages are sent to the actors, how actors expedite the processing of these control messages, and how they save and load their states during pausing and resuming, respectively. In Section 5 we study how to support conditional breakpoints in Amber, and present solutions for enforcing local conditional breakpoints (which can be checked by actors individually) and global conditional breakpoints (checked by the actors collaboratively in a distributed environment). In Section 6, we discuss challenges in supporting fault tolerance in Amber and present a technique to achieve it. In Section 7 we present the Amber implementation on top of the Orleans system [31], and report an experimental evaluation using real data sets on computing clusters to show its high performance and usability.

1.1 Related Work

Spark-based debugging. Titian [21] is a library that enables high speed data provenance in Spark. BigSift [16] is another provenance-based approach for finding input data responsible for producing erroneous results. It redefines provenance rules to prune input records irrelevant to given faulty output records before applying delta debugging [43]. BigDebug [15] uses the concept of *simulated breakpoint* in Spark execution. A simulated breakpoint needs to be preset before the execution starts, and cannot be added or changed during the execution. Furthermore, after reaching a simulated breakpoint, the results computed till then are materialized, but the computation still continues. If the user makes changes to the workflow (such as modifying a filter condition) after the simulated breakpoint, the existing execution is cancelled, causing computing resources to be wasted. In addition, the part of the workflow after the simulated breakpoint is executed again using the materialized intermediate results. Amber is different since the developer can set a breakpoint or explicitly pause the execution at any time, and the computation is truly paused.

Spark cannot support such features due to the following reason. In order for the driver (as "controller" in Amber) to send a Pause message to an executor (as "actor" in Amber) at an arbitrary user-specified time, the driver needs to send some state-change information to the executor. Spark has two ways that might be possibly used to support communication from the driver to the executor, either through a broadcast variable or using an RDD. Both are read-only to ensure deterministic computation, which is mandatory in the method used by Spark to support fault tolerance. Any state change requires a modification of the content of a broadcast variable or an RDD, and such information cannot be sent to the executor from the driver.

Workflow systems: Alteryx [3], Kepler [23], Knime [22], RapidMiner [34], and Apache Taverna [27] allow users to formulate a computation workflow using a GUI interface. They provide certain feedback to the user during data processing. These systems do not run on a computing cluster, and do not support debugging either. Texera [40] is an open-source GUI-based workflow system we are actively developing in the past three years, and Amber is a suitable backend engine. Apache Airavata [25] is a scientific workflow system supporting pausing, resuming, and monitoring. Its pause is coarse in nature since a user has to wait for an operator to completely finish processing all its data. Apache Storm [5] supports distributed computations over data streams, but does not support any low-level interactions with individual operators apart from starting and stopping the operators.

Debugging in distributed systems. When debugging a program (e.g., in C, C++, or Java) in a distributed environment, developers often use pre-execution methods such as model-checking, running experiments on a small dataset, and post-execution methods such as log analysis to identify bugs in a distributed system [8], and their limitations are already discussed above. Although query-profiling tools such as Perfopticon [28] have simplified the process of analyzing distributed query execution, their application is limited to discovering run-time bottlenecks and problematic data imbalances. StreamTrace [6] is another tool that helps developers construct correct queries by producing visualization that illustrates the behavior of queries. Such pre-execution and post-execution analysis tools cannot be used to support debugging during the execution. On the other hand, breakpoints are an effective tool to debug the run-time behavior of a program. In prior studies, global conditional breakpoints in a distributed system are defined as a set of primitive predicates such as entering a procedure, which are local to individual processes (hence can be detected independently by a process), tied together using relations (e.g., conjunction, disjunction, etc.) to form a distributed global predicate [17, 14, 26, 12]. Checking the satisfaction of a global condition given that all the primitive predicates have been detected was studied in [17, 26, 12]. Our work is different given its focus on data-oriented conditions.

Pausing/resuming in DBMS. [10] studied how to suspend and resume a query in a single-threaded pull-based engine. [4] studied how to resume online index rebuilding after a system failure. These existing approaches do not allow users to inspect the internal state after pausing.

Actor model based data processing. The use of the actor model for data processing has been explored before. For instance, S4 [29] was a platform that aimed to provide scalable stream processing using the map-reduce paradigm and actor model. Amber is different since it focuses on responsive debuggability during data processing, without compromising the scalability. Kepler [23] is a scientific workflow system using the Ptolemy II actor model implementation [33]. It is limited to a single machine and treats a grid job as an outside resource included in the workflow as an operator. Amber is different as it is a parallel run-time engine natively.

2. DEBUGGABLE DATAFLOW ENGINES

In this section, we discuss important features related to debugging the execution of a data workflow, and analyze the requirements of an engine to support these features. We then give an overview of the actor model.

2.1 Debugging Execution of Data Workflows

A data workflow (*dataflow* for short) is a directed acyclic graph (DAG) of operators. An operator is *physical* (instead of *logical*) since it specifies how its computation is done exactly, such as a hash-join operator, which is different from a ripple-join operator. We consider common relational operators as well as operators that implement user-defined functions. When running a workflow, data from sources is passed through the operators, and the results are produced from a final operator called Sink. For simplicity, we focus on the relational data model, in which data is modeled as bags of tuples, and the results generalize to other data models.

Figure 1 shows an example workflow to identify news articles related to disease outbreaks using a table of news articles (timestamp, location, content, etc.) and a table of tweets (timestamp, location, text, etc.). The KeywordSearch operator on the tweet table selects records related to disease outbreaks such as measles and zika. The next step is to find news articles published around the same time by joining them based on their timestamps (e.g., months). We then use topic modelling to classify the news articles that are indeed related to outbreaks.

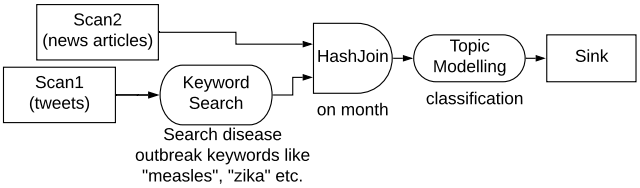

Figure 1: A workflow to anlayze disease outbreaks from tweets and news.

During the execution of a workflow, we want to allow the developer to take any of the following actions. (1) *Pausing*: stop the execution so that all operators no longer process data. (2) *Investigating operators*: check the states of each operator, and collect statistics about its behaviors, such as the number of processed records and processing time. (3) *Setting conditional breakpoints*: stop the workflow once the condition of a breakpoint is satisfied, e.g., the number of records processed by an operator goes beyond a threshold. Breakpoints can be set before or during the execution. (4) *Modifying operators*: after pausing the execution, change the logic of an operator, e.g., by modifying the keywords in KeywordSearch. (5) *Resuming*: continue the execution.

Engine requirements. A dataflow engine supporting the abovementioned debugging capabilities needs to meet the following requirements. (1) *Parallelism*: To support analytics on large amounts of data, the engine needs to allow parallel computing on a cluster. As a consequence, physically an operator can be deployed to multiple machines to run simultaneously. (2) *Supporting various messages between operators*: Developers control the execution by sending messages to operators, which should co-exist with data transferred between operators. Even if the execution is paused, each operator should still be able to respond to requests. (3) *Timely processing of control messages*: Debugging requests from the developers need to take effect quickly to improve the user experience and save computing resources. Thus control messages should be given a chance to be processed by the receiving operator without a long delay. Since processing data tuples can be time consuming, computation in an operator should be granulated, e.g., by dividing data into batches with a size parameter, so that it can handle control messages in midst of processing data.

2.2 The Actor Model

The *actor model* [20, 1] is a computing paradigm that provides concurrent units of computation called "actors." A task in this distributed paradigm is described as computation inside actors plus communication between them via messages. Every actor has a mailbox to store its received messages. After receiving a message, the actor performs three basic actions: (i) sending messages to actors (including itself); (ii) creating new actors; and (iii) modifying its

state. There are various open source implementations of the actor model such as Akka [2], CAF [9], Orleans [31], and ProtoActor [32], as well as large-scale applications using these systems such as YouScan [42], Halo 5 [18], and Tapad [38]. For instance, Halo 5 is an online video game based on Orleans that allows millions of users to play together, and each player's actions can be processed within milliseconds. There is a study to develop an actor-oriented database with support of indexing [7]. These successful use cases demonstrate the scalability of these implementations.

We use the actor model due to its several advantages. First, it is intrinsically parallel, and many implementations support efficient computing on clusters. This strength makes our system capable of supporting big data analytics. Second, the actor model simplifies concurrency control by using message passing instead of distributed shared memory. Third, the message-passing mechanism in the actor model makes it easy to support both data computation via data messages and debugging requests via control messages. Streaming control messages in the same pipeline as data messages leads to high scalability [24]. As described in Section 4.2, we can granulate the logic of operators using the actor model and thus support low-latency control-message processing.

3. AMBER SYSTEM OVERVIEW

In this section, we present the architecture of the Amber system. We discuss how it translates an operator DAG to an actor DAG and delivers messages between actors.

3.1 Architecture

Figure 2 shows the Amber architecture. The input to the system is a data workflow, i.e., a DAG of physical operators. Based on the computational complexity of an operator, the *Resource Allocator* decides the number of actors allotted to each operator. The *Actor Placement Planner* decides the placement scheme of the actors across the machines of the cluster. An operator is translated to multiple actors, and the policy of how these actors send data to each other is managed by the *Data Transfer Manager*. These modules create a DAG of actors, allocate them to the machines, and determine how actors send data. The actor DAG is deployed to the underlying actor system, which is an implementation of the actor model, such as Orleans and Akka. The execution of the actor DAG takes place in the actor system, which places the actors on their respective machines, helps send messages between them, and executes the actions of an actor when a message is received. The actor system processes the data and returns the results to the client. The *Message Delivery Manager* ensures that the communication between any two actors is reliable and follows the FIFO semantics. More details about these modules are in Section 3.2.

During the execution, a user can send requests to the system, which are converted to control messages by the *Control Signal Manager*. The actor system sends control messages to the corresponding actors, and passes the responses back to the user. The user can also specify conditional breakpoints, which are converted by the *Breakpoint Manager* to a form understandable by the engine.

3.2 Translating Operator DAG to Actor DAG

We use the example workflow of detecting disease outbreaks to show how Amber translates the operator DAG to an actor DAG, as shown in Figure 3. As in Spark, we group

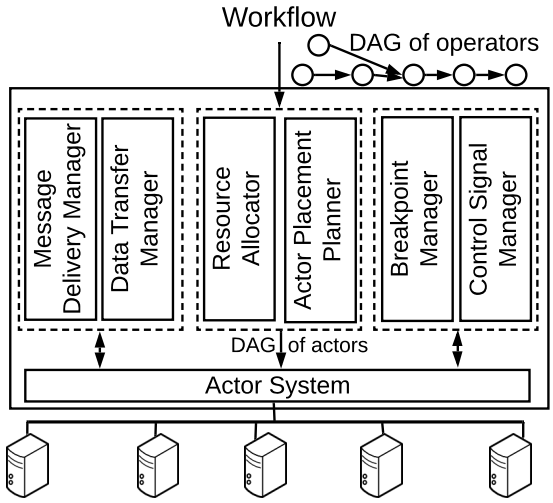

Figure 2: Amber system architecture.

a sequence of operators with no shuffling into the same stage. The operator workflow has three stages, namely {Scan1, KeywordSearch}, {Scan2}, and {HashJoin, TopicModelling, Sink}. A *controller* actor is the administrator of the entire workflow, as all control messages are first conveyed to this actor, which then routes them appropriately. The controller actor creates a *principal* actor for each operator and connects these principal actors based on the operator DAG. An edge $A \longrightarrow B$ between two actors A and B means that actor A can send messages to B. The principal actor for an operator creates multiple *worker actors*, and each of them is connected to all the worker actors of the next operators. The worker actors conduct the data-processing computation and respond to control messages. The principal actor manages all the tasks within an operator, as well as collects run-time statistics, dispatches control signals, and aggregates control responses related to its operator. Placement of workers is planned to achieve load balancing and minimizing network communication overhead and the plan is included in the actor DAG. The workers of an operator are distributed uniformly across all machines. Workers do cross-machine communication only for shuffling data at stage boundaries.

3.3 Communication between Actors

Message-delivery guarantees. Data between actors is sent as data messages, where each message includes a batch of records to reduce the communication cost. Control commands from the user are sent as control messages. These two types of messages to an actor are queued into a *single* mailbox of the actor, and processed in their arrival order. Reliability is needed to avoid data loss during the communication and FIFO is needed for some operators such as Sort. Thus, we made the communication channels between actors FIFO and exactly once. We use congestion control to regulate the rate of sending messages to avoid overwhelming a receiver actor and the network.

Data-transfer policy on an incoming edge. For each edge $A \longrightarrow B$ from operator A to operator B, the operator B has a *data-transfer policy* on this incoming edge that specifies how A workers should send data messages to B workers. If B has multiple input edges, it has a data-transfer policy for each of them. Following are a few example policies. (a)

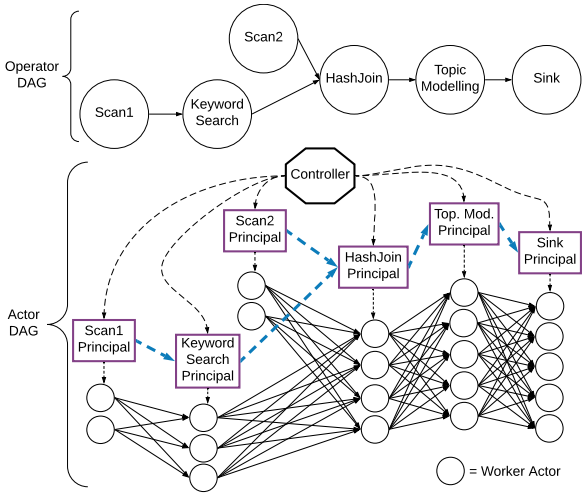

Figure 3: Translating the disease-outbreak workflow to an actor DAG. For clarity purpose, we show an edge from a principal actor to only one of its workers.

One-to-one on the same machine: An A worker sends all its data messages to a particular B worker on the same machine. (b) *Round-Robin on the same machine:* An operator A worker sends its messages to B workers on the same machine in a round-robin order. (c) *Hash-based:* Operators such as hash-based join require incoming data to be shuffled and put into specific buckets based on their hash value. An easy way to do so is to assign specific hash buckets to its workers.

4. LIFECYCLE OF JOB EXECUTION

Amber follows a staged-execution model, which means the execution follows the topological order of stages. Each worker processes one data partition and forward the results in batches to the downstream workers. Amber supports operator pipelining within a stage. In this section, we discuss the whole life cycle of the execution of a job, including how control messages are sent to the actors, how actors expedite the processing of control messages, and how each actor pauses and resumes its computation by saving and loading its states, respectively.

4.1 Sending Control Messages to Actors

When the user requests to pause the execution, the controller actor sends a control message called "Pause" to the actors. The message is sent in the following way, which is also applicable to other control messages. The controller sends a Pause message to all the principal actors, which forward the message to their workers. Due to the random delay in message delivery, the workers are paused in no particular order. For example, the source workers may be paused later than the downstream workers. Consequently, the workers may still receive data messages after being paused, and need to store them for later processing.

4.2 Expedited Processing of Control Messages

A critical requirement in Amber is fast processing of control messages in order to support real-time response from the system during debugging. Worker actors process a large number of data messages in addition to control messages. These two types of messages to a worker actor are enqueued in the same mailbox, which is a FIFO queue as specified in the actor model. Therefore, there could be a delay between the enqueuing of a control message and its processing. This delay is affected mainly by two factors, the number of enqueued messages and the computation per batch. For actor model implementations such as Akka that support priority messaging, we can expedite the processing of control messages by giving them a priority higher than data messages.

For actor model implementations that do not support priority such as Orleans, Amber solves the problem by letting each actor delegate its data processing to an external thread, called *data-processing thread* or *DP thread* for short. This thread can be viewed as an external resource used by actors to do computation and send messages to other actors. The main thread shares a queue with the DP thread to pass data messages. After receiving a data message (D_1 in Figure 4), the main thread enqueues it in the queue. The main thread offloads the data processing to the DP thread (steps (i) and (ii) in the figure). The DP thread dequeues data messages from the queue and processes them. After enqueuing a data message into the queue, the main thread is free to continue processing the next message in the mailbox. The next data messages are also stored in the queue (messages D_2 and D_3 in steps (iii) and (iv)). If the next message is a Pause message (step (v)), the main thread sets a shared variable *Paused* to true (step (vi)) to notify the DP thread. The DP thread, after seeing this new variable value, saves its states inside the worker, notifies its principal, and exits. The worker then enters a *Paused* state. The details of these actions of the DP thread will be described in Section 4.3 shortly.

While in this *Paused* state, the main thread can still receive messages in its mailbox and take necessary actions. (More details are in Section 4.4.) A received data message is stored in the internal queue (D_4 in step (vii)) because no data processing should be done. After receiving a control message, the main thread can act and respond accordingly (*Check* in step (viii)). If the control message is a Resume request, the main thread changes the *Paused* variable to false, and uses a new DP thread to resume the data processing (step (ix)). The DP thread continues processing the data messages in the internal queue, and sends produced data messages to the downstream worker (step (x)).

4.3 Pausing Data Processing

The DP thread associated with a worker actor needs to check the variable *Paused* to pause its computation so that the worker can enter the *Paused* state. One way is to check the variable after processing every data message, but this method has a long delay, especially for a large batch size and expensive operators.

Amber adopts a technique based on the observation that operators use an *iteration* model to process their tuples one by one and apply their computation logic on each tuple. Hence, the DP thread can check the variable after each iteration. If the variable becomes true, the DP thread saves necessary states of data processing in the worker actor's internal state, then simply exits. When a Resume message arrives, the main thread employs a new DP thread, which loads the saved states to resume the computation. Thus, the worker actor can respond to a Pause request quickly without introducing much overhead. The delay of checking the shared variable is mainly decided by the time of each iter-

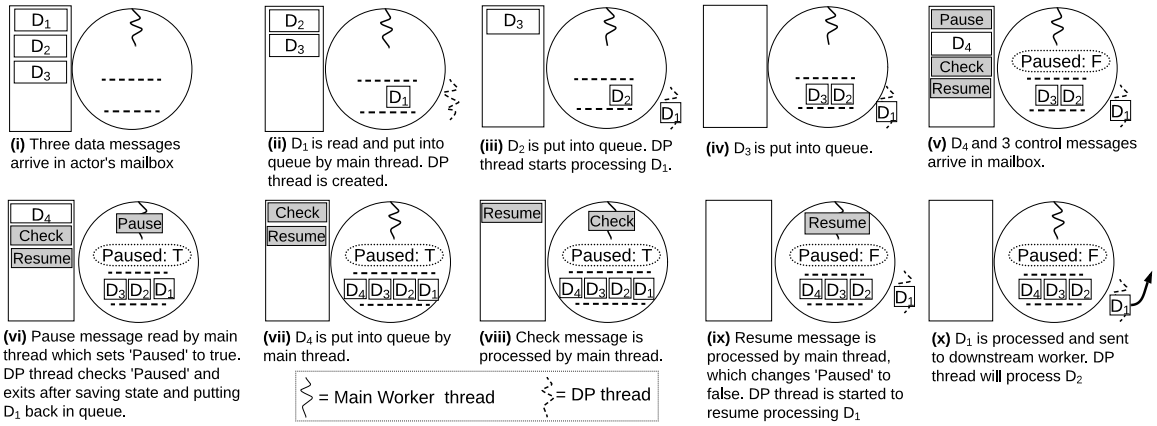

Figure 4: Processing of control and data messages by a worker. 'Paused' variable and the queue are shared between the main thread and the data processing (DP) thread. The 'Paused' shared variable is not shown in (i)-(iv) for simplicity.

ation, e.g., the time of processing one tuple. Next we use several commonly used operators as examples to show how they are implemented in Amber, and what state information is saved by a worker actor for pausing.

1. Tuple-at-a-time operators (e.g., selection, projection, and UDF operators): In a non-blocking operator, its DP thread checks the *Paused* variable after processing each tuple. When pausing, the thread needs to save the index (or offset) of the next to-be-processed tuple in the batch, called *resumption-index*.

2. Sort: Sort does not output results until receiving all its input data. We implement sort as two layers of workers. The first layer sorts its data partition and the second layer contains a single worker that merges the sorted outputs of the first-layer workers. We use this method to show how to save and load states, and the solution works for other distributed sort implementations as well. A first-layer worker does its local sort using an algorithm such as InsertionSort, MergeSort, or QuickSort. For online sorting algorithms such as InsertionSort, the DP thread saves only the resumption-index. For offline sorting algorithm such as MergeSort, the DP thread saves the state of merge sort to resume the operation later, such as the resumption index for each input chunk. The worker in the second layer merges the sorted outputs of first-layer workers to produce the final sorted results. When pausing the execution, the DP thread needs to store the resumption-index for each input batch.

3. Hash-based join. This operator consists of two phases, namely hash-building for one input table (say table R) and probing the hash table using the tuples from the other table (say table S). When a worker receives a Pause message, its DP thread can be in one of the two phases. If it is in the hash-building phase, it saves its states related to building the hash table and the resumption index for the interrupted batch of table R. If the DP thread is in the probing phase, it saves the corresponding state and the resumption index from the interrupted batch of table S.

4. GroupBy. We can implement this operator using two layers of workers. A first-layer worker does a local GroupBy for its input tuples. After that, it forwards its local aggregations to second-layer workers using a hash function on the GroupBy attribute. A second-layer worker produces final aggregated results for its own groups. When pausing the execution, the DP thread saves the resumption index of the interrupted batch and the current aggregate per group.

4.4 Responding to Messages after Pausing

After pausing the execution, the user can investigate the states of the job. For instance, she may want to know the number of tuples processed by each worker, or modify an operator, such as the constant in a selection predicate. Such requests can be implemented by sending control messages to the worker actors. Notice that even though an actor is in the *Paused* state, it can still receive and respond to messages, which is very important in debugging to support user interactions after pausing the execution (steps (vii) - (viii) in Figure 4). It is a unique capability of Amber due to the adoption of the actor model. When the user wants to resume the computation, the controller actor sends a Resume control message using the approach described above for the Pause message. Each worker actor, after receiving Resume message, uses a DP thread, to loads the saved states and continue the computation (steps (ix) and (x) in the figure).

5. CONDITIONAL BREAKPOINTS

The Amber system allows users to set breakpoints before and during the execution of a workflow in order to detect bugs and data errors. In this section, we present the semantics of conditional breakpoints, and discuss how to support two types of predicates in breakpoints.

5.1 Semantics of Conditional Breakpoints

We use an example to illustrate conditional breakpoints in Amber. Consider the workflow in Figure 1 and assume the tweets are obtained from a tab-separated text file. In this scenario, an additional operator RegexParser is required between Scan1 and KeywordSearch. This operator reads the file line-by-line and uses tab as a delimiter to convert it to multiple attribute values. In this case, both the data and the regex could have errors. For example, the `followerNum` value (i.e., number of followers of a twitter user) should always be a non-negative integer. Thus the user may want to do sanity checks on the output values of this operator. To do so, she puts a breakpoint on the output of this operator with a condition "`followerNum` < 0" When a tweet satisfies this condition, the system pauses the data processing and allows the user to investigate.

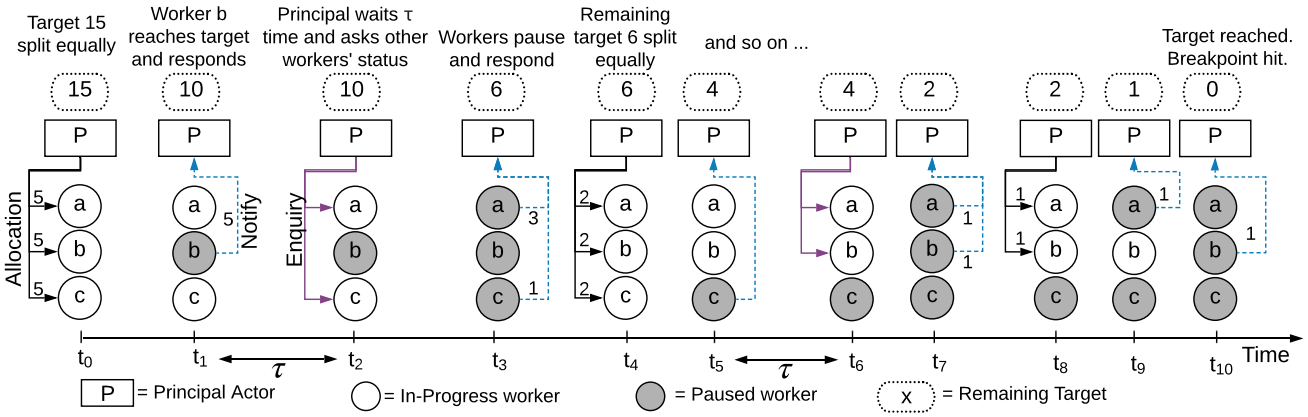

Figure 5: Evaluating a global conditional breakpoint "KeywordSearch operator producing 15 tuples." Solid lines are messages from the principal actor to workers, and dashed lines are responses from workers.

Recall that an operator can have multiple inputs and outputs. A user specifies a breakpoint on a specific output of an operator with a conditional predicate. When the predicate is satisfied, the data processing in the entire workflow should be paused. There are two types of predicates. A *local predicate* is a condition that can be satisfied by a single tuple, while a *global predicate* is a condition that should be satisfied by a set of tuples processed by multiple workers. Next we will discuss how to check these two types of predicates.

5.2 Evaluating Local Predicates

Since a local predicate can be evaluated by a single actor, such a conditional breakpoint can be detected by a worker independently. An example use case of local predicate detection is validating the schema of input data into an ML operator. For instance, the values of the 'ratings' column should always be from 1 to 5. Another use case is to pause the execution in case of an exception and show the culprit tuple to the user. Example local predicates for the workflow in Figure 1 are: 1) the `followerNum` of a tweet is negative, 2) the maximum `followerNum` among all the tweets is above 1,000. Although the second predicate is a predicate over all the tuples, it is still a local predicate since it can be checked by an actor for its tuples independently. Whenever the predicate is satisfied, the worker pauses its data processing and notifies its principal actor. Then, the principal pauses the workflow as described in Section 4.1.

5.3 Evaluating Global Predicates

A global conditional breakpoint relies on tuples processed by multiple workers of an operator, and cannot be detected by a single worker. The evaluation of such a predicate is done by the principal actor. Such predicates are valuable in scenarios where a performance metric of an operator has to be continuously monitored, e.g., the number of emails marked as spam by a Spam-Detection operator within a time window. If this metric goes above a threshold, it indicates some problem that requires attention. A few possible causes can be cyber attack, input data corruption, or model degradation. It will be helpful if the system can detect such predicates. We use two example global predicates for the workflow in Figure 1 to show how to evaluate them. G_1: the total number of tweets output by KeywordSearch is 15. G_2: the sum of `followerNum` of all tweets produced by KeywordSearch exceeds 90.

Evaluating a global `COUNT` predicate G_1. Suppose KeywordSearch has three workers. As illustrated in Figure 5, the process of evaluating G_1 consists of several steps. At time t_0, the principal actor divides the target number 15 equally among the three workers, namely a, b, and c. Each worker, after producing a tuple, increments its counter by 1. Suppose worker b is the first to produce 5 tuples. It pauses itself and notifies the principal actor (time t_1). The principal waits for a threshold time (τ) with a timer. If the other two workers respond within the time limit τ (not shown in the figure), the conditional breakpoint is hit and the principal sends a message to the controller to pause the entire workflow. Otherwise, the principal inquires each worker that has not responded, about how many records it has produced (time t_2). The figure shows the case where both workers a and c did not respond within τ. These two workers pause themselves, and respond with their number, say, 3 and 1 (time t_3). Now the remaining target number becomes $15 - 5 - (3 + 1) = 6$.

At time t_4, as before, the principal divides the new target number 6 equally, and sends a target number 2 to each worker to resume its data processing. This reassignment is necessary so that all the three workers can be resumed and the operator processes data at the maximum parallelism. Assuming worker c produces 2 tuples, it again pauses itself and notifies the principal (time t_5). The principal again waits for a threshold time after which it asks workers a and b (time t_6), who pause themselves and respond with their produced number of tuples, say 1 and 1 (time t_7). For the new remaining target of 2, the principal gives a target of 1 to each of the first two workers a and b (time t_8). Suppose worker a contacts the principal after producing one tuple (time t_9). After a while, worker b also produces a tuple, reports the same after pausing itself (time t_{10}) and the conditional breakpoint is triggered. At the end, KeywordSearch has received 15 tuples and the conditional breakpoint is hit. Notice that at time t_9, when worker a contacts the principal after producing one tuple, the principal does inquire the other workers for their tally. The reason is that there is only one tuple left to be computed, and reassigning this target to another worker will not increase parallelism.

Before the conditional breakpoint is hit, the computation can be in one of the two states. A *normal processing state* starts when the workers have been assigned their targets by the principal and ends when one of the workers completes

its target. A *synchronization state* starts when a worker completes its target and ends when the principal allocates new targets to the workers. The amount of time spent in the synchronization state depends on the timeout threshold τ and the variance of the processing speeds of the workers.

Evaluating a global `SUM` predicate G_2. The evaluation of the second predicate G_2 follows a similar process. The principal starts by dividing the target into three parts of 30 each. A unique aspect of `SUM` is that a tuple can bring the total value closer to the target by an arbitrary amount, unlike the `COUNT` example where a tuple only causes a change of 1. For example, if the current sum of `followerNum` at a worker is 24, and the next tuple has a `followerNum` value of 15, then the target of 30 will be "overshot" by 9. Therefore, it is difficult to pause the execution at the exact target.

Our goal is to minimize the amount over the target. To do so, the principal actor initially follows the same procedure as above till it gets close to the target and the overall needed value is below a threshold. The threshold can be decided based on the distribution of `followerNum` values (obtainable online). Then the principal can give the target to only one worker in order to minimize the overshot amount. For example, say the sum of `followerNum` values received by KeywordSearch till now is 80 and it needs a total of 10 more to reach the target. If it gives the three workers a target of 3, 3, and 4, and the next tuples received by the three workers have a `followerNum` value of 11, 14 and 13, respectively, the total `followerNum` sum will be 98, which is 8 more than the target of 90. Instead, if the principal gives the target of 10 to only one worker and keeps the other two paused, then even if that worker receives a tuple with a `followerNum` value such as 14, the excess is of just 4. Thus, the system pauses closer to the target.

The aforementioned methods are meant to allow the developer to pause the execution of the workflow when a conditional breakpoint is hit. As in general debuggers, it is the developer's responsibility to decide what breakpoints to set and where in order to investigate the run-time behavior of the program and find bugs. As Amber is a distributed system, the execution of multiple actors to reach a global breakpoint could be non-deterministic.

6. FAULT TOLERANCE

Fault tolerance is critical due to failures in large clusters. In traditional distributed data-processing systems such as Spark, recovery only ensures the correctness of final computed results. As we will see below, the presence of control messages in Amber poses new challenges for fault tolerance, because Amber additionally needs to ensure the recovery of control messages and their resulting states. In this section we first show why the Spark approach cannot be used, then present a solution to support fault tolerance in Amber.

6.1 Why not the Spark Approach?

Spark runs in a stage-by-stage fashion and allows checkpointing of the output of a stage. When failure happens, Spark reruns the computation of the lost data partitions from the last checkpoint using lineage information. This fault tolerance approach cannot be adopted in Amber for two reasons. Firstly, the computation of each data partition in Spark is fully independent, which allows Spark to recover only the failed partitions. In contrast, Amber has execution dependencies among workers of an operator. For example, the principal can split the global predicate in a breakpoint into multiple target numbers, which can be adjusted dynamically for each worker (see Section 5.3). If Amber naively re-runs the computation of failed data partitions, the assigned intermediate target values are lost, which will lead to an incorrect detection of the global predicate. Secondly, a control message can alter the state of a worker, and in case of failure, Amber needs to recover the worker to the same consistent state. For example, suppose before a failure happens, the worker is paused when processing the 10^{th} record in the 1^{st} data message, and the user has already seen the corresponding state of the operator of this worker, such as the number of records processed so far. After recovery, in order for the user to see the same operator state, we need to recover this worker to its state before the failure. If we were to use the Spark fault tolerance approach, this worker will not pause at all, since this approach only reruns the computation without considering the control messages. One way to support fault tolerance in Amber is using the Chandy-Lamport algorithm [11], which records all the in-transit data in a snapshot. This approach is not efficient since it can generate a large amount of checkpointed data.

6.2 Supporting Fault Tolerance in Amber

Next we develop a technique to support fault tolerance in Amber based on the following realistic assumptions. (A1) We treat the controller and the principal actors as a single unit (called "coordinator"), which is placed on the same machine. (A2) Workers only exchange data messages, not control messages. (A3) For each worker, both its computation logic and response to a control message are deterministic, as assumed by many other data-processing systems. Our fault tolerance technique consists of two parts: 1) checkpointing of data produced after each stage, and 2) logging control messages and their arrival order relative to data messages. Recovery works by restarting the computation of the failed data partitions from the last checkpoint and replaying the control messages by injecting them in the original order relative to data messages.

We use an example to explain this technique. Figure 6 illustrates the logging process of control messages before a failure of a worker (steps (i)-(iii)). A Pause message arrives at the worker after a data message with a sequence number 8 (step (i)). In step (ii), the main thread sees the Pause message and saves the sequence number 8. The main thread alters its internal state by setting the shared variable *Paused* to true. The DP thread observes the variable change after processing the 34^{th} tuple in the 6^{th} data message and notifies the main thread. In step (iii), the main thread sends the following log record to the coordinator:

```
(Pause; <Main:  8>; <DP: (6, 34)>)
```

The record includes the content of the control message (Pause), the sequence number ('8') of the last data message of the main thread when it received this control message, and the iteration status of the DP thread when it saw the shared-variable changed caused by this control message. The iteration status includes the sequence number ('6') of the currently processed data message and the index ('34') of the last processed tuple in the message. After receiving this record, the coordinator stores it in a data structure called *control-replay log* for this worker.

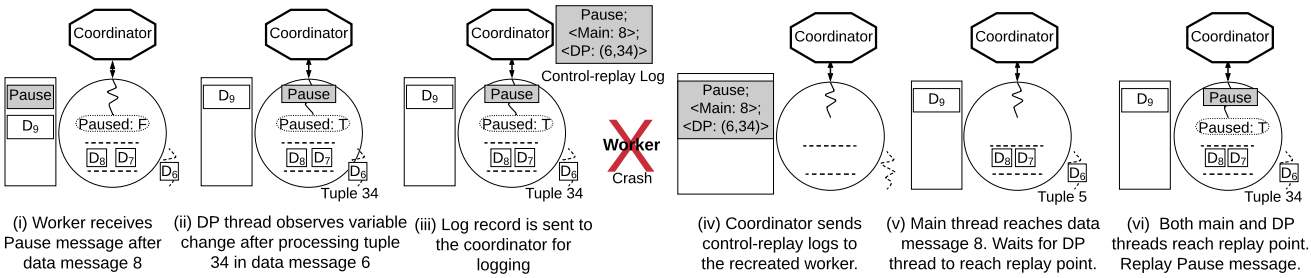

Figure 6: Fault Tolerance in Amber: logging control messages and recovery.

Suppose a machine failure happens, which causes the data partition of this worker to be lost. During recovery, the coordinator recreates all the workers of the failed partition and sends their control-replay log records respectively. During this period, the coordinator holds new control messages for each recreated worker until the worker has replayed all its control-replay log records. Each recreated worker reruns the computation of the failed partition from the last checkpoint. Since the computation is deterministic (assumption A3), a re-run of these recreated workers leads to the same content and sequence numbers of data messages received by each worker. Now let us consider the recreated worker corresponding to the aforementioned worker. Steps (iv)-(vi) in the figure show the recovery process of this worker. In step (iv), it receives its control-replay log from the coordinator. Intuitively, the main thread and DP thread of this worker continue processing their received data messages until both of them reach the control-replay point as specified in the log. Specifically, after receiving a data message D, the main thread checks the sequence number of D, denoted $S(D)$. If $S(D) < 8$, the main thread processes this message normally as before. If $S(D) = 8$, it processes D, then waits to synchronize with the DP thread (step (v)). Similarly, when the DP thread processes the tuples in a message, it handles those tuples "before" $(6, 34)$ (i.e., tuple 34 in message 6). After processing this tuple, it will synchronize with the main thread. After the synchronization, the control message is replayed by the main thread as if this message were just received (step (vi)).

There can be a case where a worker failed before being able to respond to a control message from the coordinator. For example, suppose the worker failed after step (ii) in Figure 6, before it sends the log record to the coordinator. The coordinator marks the processing of this Pause message as *incomplete*. During recovery, the coordinator first allows the recreated worker to fully replay its existing control-replay log records. The coordinator then retries sending this Pause message to the worker. Consequently, the worker can pause at a tuple different from the one before the failure. Fault tolerance is still valid because the processing of the Pause message was incomplete, and the user never saw its effect.

Amber's fault tolerance approach incurs little overhead on execution because it only saves control messages and control-replay log, which have a much smaller size compared to data messages. To deal with the case of coordinator failures, we can use write-ahead logging or use backup coordinators to replicate the states of the coordinator. Notice that for an operator with multiple data inputs such as Join and Union, to satisfy assumption A3, they need to provide an ordering guarantee across their inputs, and the sequence numbers of each input will be maintained and recorded separately. If failure happens during recovery, the coordinator can simply restart the recovery procedure, which is idempotent.

7. EXPERIMENTS

In this section, we present an experimental evaluation of the Amber system using real data sets on clusters.

7.1 System Implementation and Setting

Data and Workflows. We used three real datasets, namely TPC-H, tweets, and New York taxi events. For the TPC-H benchmark [41], we varied the scale factor to produce data of different sizes. Based on the TPC-H queries 1 and 13 we constructed two workflows, shown as W_1 and W_2 in Figure 7. Note that the Scan operators of W_1 and W_2, had a built-in projection to read only the columns being used by the operators later. This improvement was used in the experiments for both Amber and Spark. The second dataset included 100M tweets in the US, on which we did sentiment analysis using an ML-based, computationally expensive operator. The third dataset included New York City Yellow taxi trips (about 210 GB), and each record had information about a trip, including its pick-up and drop-off geo-locations, times, trip distance, payment method, and fare [39].

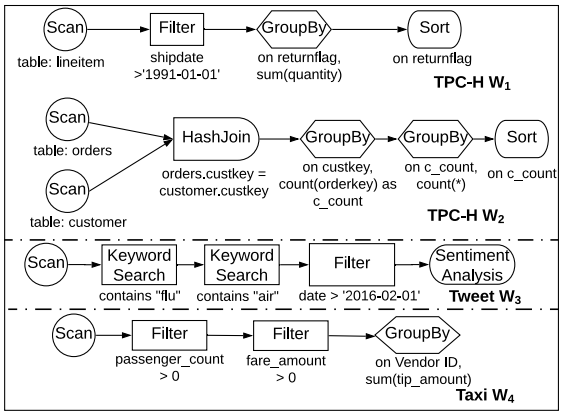

Figure 7: Workflows used in the experiments.

Experiment Setting. All the experiments were conducted on Google Cloud Platform (GCP). The data was stored in an HDFS file system on a storage cluster of 51 `n1-highmem-4` machines, each with 4 vCPU's, 26 GB memory, and 500GB

standard persistent disk space (HDD's). The execution of a workflow was done on a *separate* processing cluster of 101 machines with the same type. The storage cluster and processing cluster were running Debian GNU/Linux 9.9 (stretch) and Debian GNU/Linux 9.11 (stretch) operating system respectively. The batch size used in data messages was 400 unless otherwise stated. Data checkpointing was disabled by default in all experiments, except the experiment concerning fault tolerance (Section 7.7). Out of the 101 machines, we used one just for the controller and principal actors of the operators, and the remaining 100 for data processing. When reporting the number of computing machines, we only included the number of data-processing machines.

We implemented Amber in C# on top of Orleans (version 2.4.2), running on the .Net core run-time (version 3.0). The operators were implemented as discussed in Section 4.3, and the workers of each operator were assigned uniformly across multiple machines. For example, if a Scan operator had 10 workers and the processing cluster had 10 machines, then each machine had a single Scan worker.

7.2 Scaleup Evaluation

We evaluated the scaleup of Amber using the TCP-H data. We started with a data set of 10GB processed by 1 machine (4 cores), and gradually increased both the data size in the storage cluster and the machine number in the processing cluster linearly to 1TB processed by 100 machines (400 cores). For both workflows W_1 and W_2, the early operators did most of the work, leaving very few (less than 50) tuples for the final Sort operator. Therefore, we allocated 2 workers for each operator on each machine, except Sort that was allocated 1 worker on each machine. The GroupBy operator had two layers (Section 4.3). The first layer was allocated 2 workers on each machine and the second layer was allocated 1 worker on each machine.

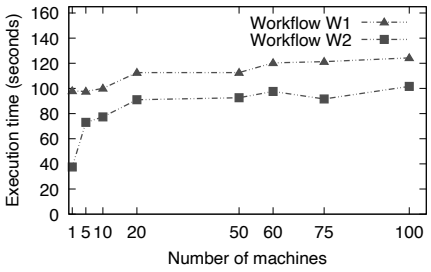

Figure 8: Scaleup of TPC-H workflows W_1 and W_2

Figure 8 shows the running time for workflow W_1. For the 10GB data processed by 1 machine, the total time was around 98s. When we increased the data size and the cluster size gradually, the time increased slightly. For the 1TB data processed by 100 machines, the total time was around 124.3s. Figure 8 also shows the results for W_2. For the 10GB data processed by 1 machine, the total time was around 37.5s. When we increased the data size and the cluster size gradually, the time increased at a faster rate than W_1 because of the intrinsic quadratic complexity of Join. It took 101.6s for 100 machines to process 1TB data.

7.3 Speedup Evaluation

To evaluate the speedup of Amber, we measured the time taken to execute workflows W_1 and W_2 on the 50GB data using 1 computing machine initially and gradually increased the number of computing machines to 100.

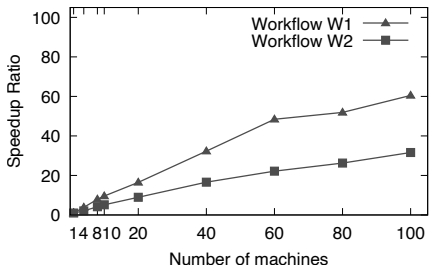

Figure 9: Speedup for TPC-H workflows W_1 and W_2

Figure 9 shows the speedup for workflow W_1. For the 50GB data processed by 1 machine, the total time was around 484.5s. When we increased the number of machines gradually, the time decreased. The time was about 10s when using 60 machines, with a speedup ratio of 48.4. When we increased the number of machines further to 80 and then 100, the total time taken did not decrease at the same rate. For 100 machines, the total time taken was 8s (with a speedup ratio of 60.5). This result was due to the fact that the total data to be processed was only 50GB and the machines were not fully utilized. Figure 9 also shows the results for W_2. Its speedup was sub-linear due to the intrinsic quadratic complexity of Join. Increasing the number machines from 60 to 100 yielded little performance gain due to the increased communication cost.

7.4 Time to Pause Execution

We used Pause and Resume as examples to evaluate the time taken to process a control message while a workflow is running on a cluster. We did the experiment with the similar setting as the scaleup experiments. Each execution was interrupted 8 times by sending a Pause then a Resume message, before its completion.

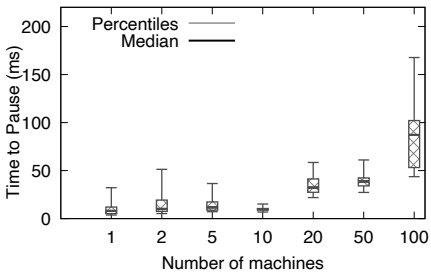

Figure 10: Time taken to pause the execution while scaling up TPC-H workflow W_1. 1^{st} percentile, 1^{st} quartile, median, 3^{rd} quartile, and 99^{th} percentile are shown.

Figure 10 and Figure 11 show the candlestick chart with 1^{st} percentile, 1^{st} quartile, median, 3^{rd} quartile, and 99^{th} percentile pause times for W_1 and W_2 respectively. All the times were less than 1 second. The time to pause W_2 was relatively more than that of W_1 due to the high number of data messages received by the Join operator, resulting in more time to reach the Pause message. The time to resume each workflow was also in milliseconds. The time to pause increased with the number of machines due to the inherent increase in the communication cost and higher number of

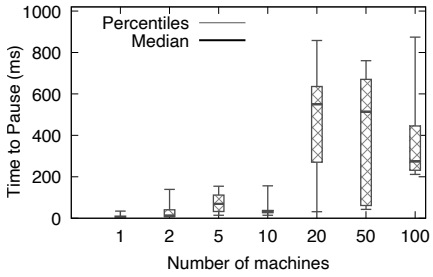

Figure 11: Time taken to pause the execution while scaling up TPC-H workflows W_2. 1^{st} percentile, 1^{st} quartile, median, 3^{rd} quartile, and 99^{th} percentile are shown.

data messages being received by the operators. These results show that control messages in Amber can be handled quickly during the processing of a large amount of data. Note that time to pause depends on the delay of checking the shared variable Pause by the DP thread, which is approximately equal to the time required by the operator to process a single tuple. The delay in the relational operators was in milliseconds. For complex operators that need more time to process one tuple such as ML operators, the time to pause could be higher.

7.5 Effect of Worker Number

A unique feature of Amber is that different operators can have different numbers of workers. We used workflow W_3 on the tweet data set to evaluate the effect of the number of workers allocated to computationally expensive operators. It included an SentimentAnalysis operator, which was based on the CognitiveRocket package [35] and needed about 4 seconds to process each tuple. We used it as an example of expensive ML operators. The workflow took 100M tweets as the input and first applied KeywordSearch and Filter operators. The number of tweets going into the SentimentAnalysis operator was 1,578. We varied the total number of workers allotted to the SentimentAnalysis operator and measured its effect on the total running time. The total number of workers for all other operators was fixed at 10. The workflow was run on a cluster of 10 machines with a batch size of 25 for data from Filter to SentimentAnalysis operator. We used a smaller batch size because we only had 1,578 tuples to be distributed among sentiment analysis workers.

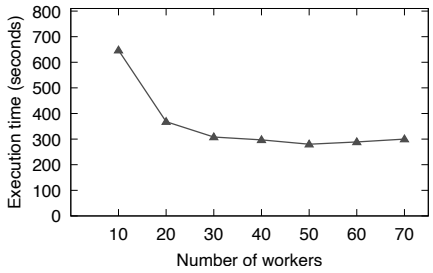

Figure 12: Changing the worker number of the SentimentAnalysis operator in workflow W_3 for tweets.

Figure 12 shows the results.The running time when the operator had 10 workers was 647s. When we doubled the number of workers, the execution time reduced to 368s. The rate of decrease declined as the number of workers was further increased. When the number of workers was increased above 50, the execution time even started increasing, because data got distributed among many workers which competed for CPU, thus increasing the overhead of context switching. Thus, workers took more time to finish their task.

Dynamic resource allocation. ML operators such as SentimentAnalysis can become a bottleneck in workflows due to their slow speed of computation and extra resources needed under peak load conditions. We implemented the technique of dynamic resource allocation as suggested in [24] to allocate extra machines to the SentimentAnalysis operator *during* the execution, and evaluated the performance gain. First, we ran W_3 on a cluster of 6 machines, with each operator having 1 worker per machine, except SentimentAnalysis that had 5 workers per machine, and the total time to run W_3 was 422s. Then, we modified the setting by adding one more machine to the cluster every minute and allocating 5 SentimentAnalysis workers on the newly added machine. The total time to run W_3 reduced to 407s. This reduction of 15s was feasible because of Amber's capability to add more computing resources dynamically.

7.6 Conditional Breakpoint Evaluation

For the TPC-H workflow W_1 running on 10 machines, we used 119M tuples and set a conditional breakpoint on the output of the Filter operator to pause the workflow after this operator produced 100M tuples. We varied the timeout threshold (τ) used by the principal from 0ms to 5s, and measured the time in the normal processing state and the time in the synchronization state, as discussed in Section 5.3. Figure 13 shows the results. The normal processing time was about 30s. The synchronization time was relatively small. When τ was high, the total synchronization time was around 2.15s. When we decreased τ, the synchronization time decreased. The overall time decreased with decreasing τ since we had more data parallelism. The best setting was when τ was a few milliseconds.

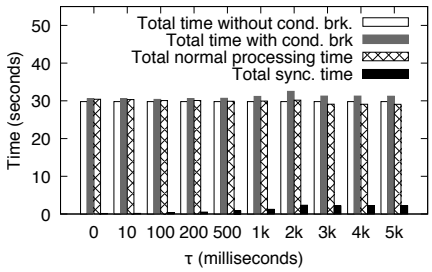

Figure 13: Conditional breakpoint: running time versus principal's waiting threshold τ.

To evaluate the overhead of conditional breakpoints, we measured the total time needed by Filter operator to produce 100M tuples when the input had 119M tuples and no conditional breakpoint was set. Figure 13 shows that the time taken by the Filter operator to produce the same 100 million tuples was about 29.8s, which was close to the overall time taken with the conditional breakpoint.

7.7 Performance Comparison with Spark

We compared the performance of Amber with Apache Spark using TPC-H W_1 and W_2. Data checkpointing was disabled for Spark. The scaleup experiment settings were

re-used for Spark. Similar to Amber, we put the Spark's driver on one dedicated machine and allowed its executors to run on the other 100 machines. We used two Spark API's, namely the DataFrame API on top of the Spark SQL engine, and the RDD API, which is its primary user-facing API [37]. The RDD API is a more general API that supports user-defined data structures and transformations. The DataFrame API is a faster SQL-based API because of many optimizations, such as binary data formats, fast serialization, and code generation [36].

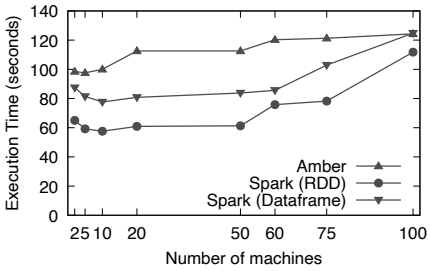

Figure 14: Scaleup for Amber and Spark for W_1.

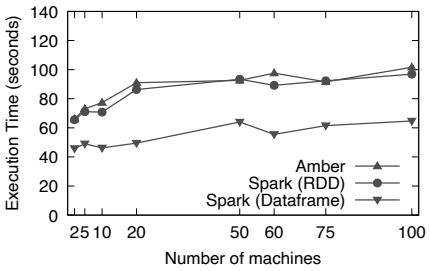

Figure 15: Scaleup for Amber and Spark for W_2.

Figures 14 and 15 show the results for TPC-H W_1 and W_2, respectively. Amber achieved a performance comparable to Spark's DataFrame API and even more comparable to the RDD API for W_2. The performance gain of Spark's DataFrame API can be attributed to its optimizations discussed earlier. However, to our surprise, Spark's RDD API outperformed Spark's DataFrame API for W_1. Amber performance remained quite comparable to both the API's for W_1 too. We also compared Amber and Spark (DataFrame API) using the Taxi workflow W_4 on 10 machines. Spark took 442s, while Amber took 470s.

7.8 Fault Tolerance in Amber and Spark

To evaluate the overhead of supporting fault tolerance, we turned on data checkpointing in Amber and Spark to write checkpointed data to a remote HDFS. In Amber, a worker created a separate file for each hash partition. In contrast, Spark consolidates its checkpoint data into HDFS block-sized files of 128MB. With data checkpointing on, we scaled the execution of W_2 on both systems from 2 machines (20GB data) to 20 machines (200GB data) and let the workflow run to completion. We chose W_2 because it has more number of stages than W_1. In order to reduce the number of produced files, we used only 1 worker per operator per machine for Amber. We used the DataFrame API of Spark as it was faster than the RDD API for W_2. Figure 16 shows the execution times. Amber's data checkpointing initially performed better than Spark. For a higher number of machines, Amber took more time to complete the execution due to the quadratic increase in the number of files. For instance, for the 20-machine case, Amber produced 400 files (20 workers, each producing 20 partitions) at the end of each stage, while Spark produced only 66 files.

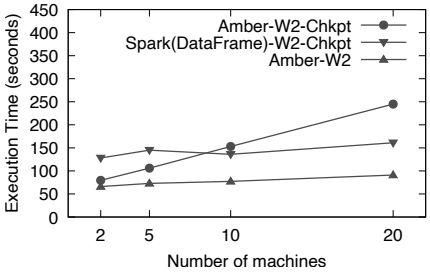

Figure 16: Data-checkpointing overhead for Amber and Spark while executing W_2. The time for Amber with data checkpointing disabled and time for Spark and Amber with data checkpointing enabled are shown.

We evaluated crash recovery for Amber. When running W_2 on 10 machines (100GB data), after the Join operator ran for 5s, we simulated a crash by killing the workers of one data partition. Amber took 176s in total (including crash and recovery) to run to completion, which was comparable to the case where there was no failure (153s). We also evaluated recovery of control messages in Amber. We paused the workflow after it entered the Join stage for 10s and then simulated a crash. Amber took 6s to recreate actors, and 10s of recomputation to recover to the same Paused state.

Summary: The experiments showed that the Orleans-based Amber implementation can process control messages quickly and support conditional breakpoints with a low overhead. It achieved a high performance (both scaleup and speedup) comparable to Spark. The capability of supporting dynamic resource allocation during the execution achieved a better performance. Given its young age compared to other data-processing systems, Amber has the potential to achieve an even better performance.

8. CONCLUSIONS

In this paper we presented a system called Amber that supports powerful and responsive debugging during the execution of a dataflow. We presented its overall system architecture based on the actor model, studied how to translate an operator DAG to an actor DAG that can run on a computing cluster. We described the whole lifecycle of the execution of a workflow, including how control messages are sent to actors, how to expedite the processing of these control messages, and how to pause and resume the computation of each actor. We studied how to support conditional breakpoints, and presented solutions for enforcing local conditional breakpoints and global conditional breakpoints. We developed a technique to support fault tolerance in Amber, which is more challenging due to the presence of control messages. We implemented Amber on top of Orleans, and presented an extensive experimental evaluation to show its high usability and performance comparable to Spark.

Acknowledgements: We thank the Microsoft Orleans team and Dr. Phil Bernstein for answering questions related to Orleans, Yuran Yan for helping us evaluate Apache Spark, and Google Cloud Platform. This work was partially supported by NSF award IIS-1745673.

9. REFERENCES

[1] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, Massachusetts Inst Of Tech Cambridge Artificial Intelligence Lab, 1985.

[2] Akka Website, https://akka.io/.

[3] Alteryx Website, https://www.alteryx.com/.

[4] P. Antonopoulos, H. Kodavalla, A. Tran, N. Upreti, C. Shah, and M. Sztajno. Resumable online index rebuild in SQL server. *PVLDB*, 10(12):1742–1753, 2017.

[5] Apache Storm, http://storm.apache.org/.

[6] L. Battle, D. Fisher, R. DeLine, M. Barnett, B. Chandramouli, and J. Goldstein. Making sense of temporal queries with interactive visualization. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, San Jose, CA, USA, May 7-12, 2016*, pages 5433–5443, 2016.

[7] P. A. Bernstein, M. Dashti, T. Kiefer, and D. Maier. Indexing in an actor-oriented database. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[8] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems. *Commun. ACM*, 59(8):32–37, 2016.

[9] CAF Website, https://actor-framework.org/.

[10] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query suspend and resume. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 557–568, 2007.

[11] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[12] D. Dao, J. Albrecht, C. Killian, and A. Vahdat. Live debugging of distributed systems. In *International Conference on Compiler Construction*, pages 94–108. Springer, 2009.

[13] D. Fisher, R. DeLine, M. Czerwinski, and S. M. Drucker. Interactions with big data analytics. *Interactions*, 19(3):50–59, 2012.

[14] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*, pages 134–141, 1990.

[15] M. A. Gulzar, M. Interlandi, T. Condie, and M. Kim. Debugging big data analytics in spark with *BigDebug*. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1627–1630, 2017.

[16] M. A. Gulzar, M. Interlandi, X. Han, M. Li, T. Condie, and M. Kim. Automated debugging in data-intensive scalable computing. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 520–534, 2017.

[17] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *[1988] Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences. Volume II: Software track*, volume 2, pages 166–175. IEEE, 1988.

[18] Halo Website, https://www.halowaypoint.com/en-us.

[19] C. Herath, F. Liu, S. Marru, L. Gunathilake, M. Sosonkina, J. P. Vary, P. Maris, and M. E. Pierce. Web service andworkflow abstractions to large scale nuclear physics calculations. In *2012 IEEE Ninth International Conference on Services Computing, Honolulu, HI, USA, June 24-29, 2012*, pages 703–710, 2012.

[20] C. Hewitt, P. B. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245, 1973.

[21] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. D. Millstein, and T. Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[22] Knime Website, https://www.knime.com/.

[23] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[24] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao. Chi: A scalable and programmable control plane for distributed stream processing systems. *PVLDB*, 11(10):1303–1316, 2018.

[25] S. Marru, L. Gunathilake, C. Herath, P. Tangchaisin, M. E. Pierce, C. Mattmann, R. Singh, T. Gunarathne, E. Chinthaka, R. Gardler, A. Slominski, A. Douma, S. Perera, and S. Weerawarana. Apache airavata: a framework for distributed applications and computational workflows. In *Proceedings of the 2011 ACM SC Workshop on Gateway Computing Environments, GCE 2011, Seattle, WA, USA, November 18, 2011*, pages 21–28, 2011.

[26] B. P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, California, USA, June 13-17, 1988*, pages 316–323, 1988.

[27] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. Taverna, reloaded. In *International conference on scientific and statistical database management*, pages 471–481. Springer, 2010.

[28] D. Moritz, D. Halperin, B. Howe, and J. Heer. Perfopticon: Visual query analysis for distributed databases. *Comput. Graph. Forum*, 34(3):71–80, 2015.

[29] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: distributed stream computing platform. In *ICDMW 2010, The 10th IEEE International Conference on Data Mining Workshops, Sydney, Australia, 13 December 2010*, pages 170–177, 2010.

[30] C. Olston and B. Reed. Inspector gadget: a framework for custom monitoring and debugging of distributed dataflows. In *Proceedings of the ACM*

SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011, pages 1221–1224, 2011.

[31] Orleans Website, https://dotnet.github.io/orleans/.

[32] ProtoActor Website, http://proto.actor.

[33] Ptolemy II Website, https://ptolemy.berkeley.edu/ptolemyII/ptII11.0/index.htm.

[34] RapidMiner Website, https://rapidminer.com/.

[35] Sentiment Analysis operator for .Net, https://github.com/arafattehsin/CognitiveRocket/tree/master/CognitiveLibrary/SentimentAnalyzer.

[36] Spark DataFrame API optimizations in Projet Tungsten, https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[37] Differences between Spark RDD API and Dataframe API, https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-data.html.

[38] Tapad Website, https://www.tapad.com.

[39] NYC TLC Trip Record Data, https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[40] Texera Website, https://github.com/Texera/texera.

[41] TPC-H Website, http://www.tpc.org/tpch/.

[42] YouScan Website, https://youscan.io/en/.

[43] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.