Consistent Regions: Guaranteed Tuple Processing in IBM Streams

Gabriela Jacques-Silva[♠*], Fang Zheng[♠], Daniel Debrunner[‡], Kun-Lung Wu[♠], Victor Dogaru[‡], Eric Johnson[‡], Michael Spicer[‡], Ahmet Erdem Sarıyüce[⋄†]

[♠]IBM T. J. Watson Research Center, [‡]IBM Analytics Platform, [⋄]Sandia National Labs

Abstract

Guaranteed tuple processing has become critically important for many streaming applications. This paper describes how we enabled IBM Streams, an enterprise-grade stream processing system, to provide data processing guarantees. Our solution goes from language-level abstractions to a runtime protocol. As a result, with a couple of simple annotations at the source code level, IBM Streams developers can define *consistent regions*, allowing any subgraph of their streaming application to achieve guaranteed tuple processing. At runtime, a consistent region periodically executes a variation of the Chandy-Lamport snapshot algorithm to establish a consistent global state for that region. The coupling of consistent states with data replay enables guaranteed tuple processing.

1. INTRODUCTION

Stream processing systems have become one of the basic building blocks for tackling big data problems [3, 16, 22, 25]. With such systems, developers can write their applications as explicit data flow graphs, where each node of the graph is a *stream operator* and each edge is a *stream connection*. Stream operators can generate *tuples* from external sources, or apply transformations on incoming tuples and further submit them downstream via their output stream connections. A stream processing platform is responsible for deploying and managing the execution of such a data flow graph in a distributed environment.

Most streaming systems focus primarily on achieving high-throughput and low latency. Moreover, as many streaming applications are able to handle approximate results, several of the fault tolerance techniques designed for streaming applications favor the timeliness of the output over the guaranteed processing of every tuple [11, 17, 19, 20, 22, 27]. We call such techniques *partially fault-tolerant*. However, as developers are applying the streaming paradigm to more application domains, guaranteed tuple processing has become a primary concern. For example, in many health care and telecommunication applications, it is important that every tuple is processed.

In this paper, we describe how we enabled IBM Streams, an enterprise-grade stream processing system originally designed to be partially fault-tolerant, to provide guaranteed tuple processing. Our technique, called *consistent regions*, enables developers to select subsets of streaming operators in an application to process tuples at-least-once, and in some cases exactly-once. A *consistent region* consists of a subgraph of operators in a streaming application. At runtime, the region executes a variation of the Chandy-Lamport algorithm [14] for establishing a consistent distributed snapshot for the operators' state. The periodic establishment of consistent states coupled with data replay provides at-least-once processing and enables exactly-once semantics with additional constraints.

In IBM Streams, applications can be non-deterministic, have arbitrary topologies (including cycles), have multi-threaded stream operators and implement any data processing logic. In this scenario, the distributed snapshot algorithm is a good fit, as it does not rely on any of these characteristics for correctness.

Although conceptually simple to understand, implementing the distributed snapshot protocol in an enterprise-grade system is a significant challenge. From a streaming platform perspective, we must consider (i) the performance impact of the protocol on applications that do not require guaranteed processing, (ii) the code complexity of the implementation, (iii) existing framework concepts and APIs, and (iv) the ease of adapting legacy applications. From an application developer's perspective, the primary concerns are to easily configure the application to use the protocol, and the overall overhead of the solution.

To tackle the platform challenges, we have done the following: (i) minimal changes to the main tuple processing path, so that applications that do not need guaranteed processing can run at full speed; (ii) a variation of the distributed snapshot protocol by adding a *drain* stage, which ensures that all in-flight tuples are processed before establishing a consistent state. This stage forces any in-flight tuples for a stream connection (channel state) to be processed and reflected in the operator state (process state). This eliminates the complexity and cost of persisting in-flight data and playing this data after the checkpoint is complete; (iii) leveraging the existing concept of a *stream punctuation* to represent the *tokens* used in the original distributed snapshot protocol.

[*]This work was done while the author was an RSM at IBM.
[†]This work was done while the author was an intern at IBM.

Proceedings of the VLDB Endowment, Vol. 9, No. 13

This enables the reuse of all the machinery for punctuation processing and to more easily integrate the protocol into the platform; and (iv) addition of a state management interface, which enables operator state persistence, and a new synchronization primitive, which allows multi-threaded operators to establish logical boundaries on the code in which it is safe to establish a consistent state. With these new APIs, operators can be adapted to work with the protocol.

For application developers, we offer two language-level abstractions, so that with simple annotations at the source-code level, application subgraphs can achieve guaranteed tuple processing. The first annotation adds an application subgraph to a consistent region. The second one removes operators that do not require guaranteed tuple processing from a region. As a result, only operators in the region are required to coordinate during the establishment of a consistent state, reducing the overall cost of the protocol. The overall overhead of consistent regions for an application is proportional to how often the protocol executes. This is configurable directly by the developer by parameterizing the annotation with a period or an operator logical boundary (e.g., at every fully processed file). Allowing fine-grained selection also gives developers flexibility to specify different policies for different subgraphs of the application. For example, subgraphs with large in-memory state can be checkpointed less often than other parts of the application.

The key contributions of this paper are: (i) a variation of the distributed snapshot algorithm for streaming applications which includes a *drain* stage to force any channel state to be reflected as operator state. While running the drain stage, the application is still doing useful work towards its progress; (ii) language-level abstractions that enable an application to do fine-grained selection of the subgraphs that run with a tuple processing guarantee. This reduces the number of operators required to do coordination during the establishment of a consistent state; (iii) runtime APIs that expose the stages of the protocol to a streaming operator and enable multi-threaded operators to participate in consistent regions. We have adapted over 70 commonly-used operators to use the APIs so that they can be used in consistent regions. To the best of our knowledge, this is the first fault tolerance solution that allows fine-grained configuration of tuple processing guarantee within a streaming application, supports non-deterministic processing logic, and handles multi-threaded operators.

2. OVERVIEW

As described in Section 1, consistent regions provide both at-least-once and exactly-once guaranteed tuple processing. We define the guarantee in terms of the application output. When referring to *at-least-once*, we mean that the application produces correctly computed output – as if no failures had happened, but tuples may show up more than once on the application output. For *exactly-once*, we mean that the application produces correctly computed output and that tuples do not appear duplicated in the output, regardless of failures. The application output is always correct because we implement a variant of the Chandy-Lamport algorithm for establishing a distributed snapshot [14], where operator states are persisted by default.

The establishment of a consistent state has two stages: the *drain* stage and the *checkpoint* stage. During a drain, operators in a region are allowed to submit any pending tuples (e.g., buffered data). Once the drain finishes, no new tuple submission is allowed. After a drain, the checkpoint stage starts. In this stage, the operator persists its state to the checkpoint backend store. After all operators in a consistent region complete both stages, the consistent state is successfully established.

To guarantee that all in-flight tuples are either fully processed by the region or reflected in operator state, the establishment of a consistent state happens in topological order of the consistent region subgraph. The process begins at the *start operators* of a consistent region. In general, these are the source operators (operators without any incoming streams). Once both the drain and checkpoint stages complete, an operator submit a *drain marker* to all its output streams[1]. The drain marker is a special punctuation that flows through the stream connections of the consistent region[2]. Any tuple submitted prior to or during the drain stage is guaranteed to be processed before the completion of the current consistent state. Similar to the original distributed snapshot protocol [14], the correctness of our protocol relies on all stream connections enforcing FIFO order.

When a non-start operator in the region receives a drain marker from all of its incoming streams, the drain stage starts followed by the checkpoint stage. Drain markers are then forwarded to its output streams. For each operator, the checkpoint stage happens only after all input streams of the operator are fully processed. This ensures that the stream connections are empty and that no persistence of channel state is required. Once all operators in the region finish both the drain and checkpoint stages, a central *consistent region controller* notifies the start operators that they can resume processing new data. Note that even though the region stops processing new tuples while the consistent state is being established, useful work is still being performed while the drain stage executes. This is because tuples submitted prior to or during the drain stage are still being processed by downstream operators.

In addition to maintaining the different stages of the protocol, the consistent region controller is responsible for detecting failures. It relies on the IBM Streams platform infrastructure to continuously monitor the health of processes and connections in a consistent region. When the controller detects a failure in a region or the controller itself fails, it resets the region to the last consistent state. The restoration has a single stage called *reset*. In this stage, each operator retrieves its state from the checkpoint backend store. Once all operators in the region reset their state, the start operators of the region can start replaying tuples.

The restoration propagates a special punctuation called *reset marker* through the region to clear up the stream connections from tuples submitted prior to the reset notification. When all input stream connections of an operator are cleared, the state reset takes place. This ordering ensures that intermediate operators of a consistent region do not need to deal with duplicate tuples.

3. LANGUAGE-LEVEL ABSTRACTIONS

In IBM Streams, developers write their streaming applications using the Streams Processing Language (SPL). An

[1]Section 4.2.3 describes further protocol optimizations.

[2]Special drain marker propagation rules are used in cyclic topologies.

application is composed of operators that can consume input streams and generate output streams.

SPL has two kinds of operators. The first kind are primitive operators, which can be written in C++, Java, or SPL itself (also called Custom). Primitive operators written in C++ can leverage a code-generation framework to generate specialized code depending on the operator configuration. The second kind of operators are composite operators. Composites enable modularization and code reuse by encapsulating subgraphs of primitive and other composite operators. The SPL compiler is responsible for invoking the code-generation framework, expanding the composite operators to create the application topology, and generating the application binaries.

In SPL, an operator can have multiple input ports and multiple output ports. An input port can have multiple input stream connections. In general, when processing tuples from an input port, the operator generates a new tuple that is *submitted* to one of its output ports. These ports are called *data ports*. If the processing logic of an input port does not generate tuple submissions, the input port is called a *control port*. It is safe for the SPL compiler to close a feedback loop on a control port. Feedback loops are useful when changing the runtime behavior of an operator (e.g., a filtering condition) based on downstream processing.

During runtime, operators are deployed in *processing elements* (PEs), which run as operating system processes. Operators in the same PE communicate to each other via function calls. Optionally, the application can gain pipeline parallelism by configuring an operator input port as threaded. This results in a queue being added between two operators. The queue is read by a new thread, which picks up the tuple processing from that point of the graph onwards. Operators in different PEs communicate via TCP connections.

One of the key aspects we chose to support for consistent regions is the fine-grained selection of one or more subgraphs of an application requiring guaranteed tuple processing. This is because different parts of an application can have different requirements regarding tuple processing guarantees. For example, a streaming application can correlate banking transactions with information derived from Twitter feeds to find marketing opportunities in real time. While the application requires the subgraph processing banking transactions and doing the correlation to process every tuple, it is acceptable for the subgraph processing the Twitter stream to lose some of its tuples. As a result, there is no need to enforce the Twitter subgraph to be coordinated with the banking transactions and correlation subgraph.

Another motivation for enabling fine-grained subgraph selection is to not give false promises to an application developer. Some operators cannot easily participate in a consistent region, as it does not naturally support a checkpointing or tuple replay scheme. One example is an operator consuming data from a UDP socket and submitting it as tuples downstream. If the operator fails, it cannot replay tuples without extra machinery, as a UDP socket is not replayable. Another example is an operator that has in-memory state but is not capable of persisting its state. The SPL compiler can forbid such operators to be used in a consistent region, so that the application developer does not expect that such operators will be made consistent during runtime.

In SPL, a consistent region is specified using annotations. The `@consistent` annotation defines a consistent region, and the `@autonomous` annotation can be then used to reduce the scope of a region.

3.1 Consistent Annotation

The `@consistent` annotation is placed on an operator of the application. The SPL compiler then automatically identifies the consistent region by computing the reachability graph of the annotated operator. The annotated operator is the *start* operator of the region. All operators in the reachability graph that do not have any downstream operators are identified as the *end* of the region. When the annotation is placed on a composite operator, the compiler automatically identifies the start operators of the composite and computes the reachability graph from those operators. A consistent region can have multiple start and end operators.

An application can have multiple consistent regions. This means that each region establishes consistent states independently. Furthermore, the failure of an operator in one region does not cause the reset of operators in other regions. When annotations are placed on two different operators and their reachability graphs have a common operator, the compiler merges the two consistent regions into one.

For example, Figure 1(a) shows an application with a single consistent region. In this example, op_1 is annotated with `@consistent` and is identified by the compiler as the start operator of the region. As a result, all operators in its downstream are included in the region. As operators op_5 and op_6 have no output streams, the compiler identifies them as the end of the consistent region. Figure 1(b) shows a case in which `@consistent` is placed on two operators: op_1 and op_3. The reachability graphs of both of them have operators op_5 and op_6, so the regions are merged by the compiler.

In addition to selecting the region, the `@consistent` annotation is used to configure the runtime behavior of the region. The main parameter of the annotation is the `trigger` parameter. The trigger indicates how to start the establishment of a consistent state. It can be either *periodic* or *operator-driven*. A periodic consistent region establishes consistent states according to a specified period. This kind of region can have multiple start operators. An operator-driven region establishes consistent states according to the logical boundary specified by the start operator of the region. For example, consider an operator called DirectoryScan that scans a directory, reads its current contents, and submits a tuple with a file name attribute for every file found in the directory. This operator is frequently used together with an operator that reads the file and submits its contents downstream. When in an operator-driven consistent region, the DirectoryScan operator can choose to establish a consistent state after submitting every tuple (i.e., every file). This ensures that a consistent state is established after fully processing a file. This is a powerful abstraction, as it enables the establishment of consistent states at points that make semantic sense to the application. Currently, operator-driven regions can only have a single start operator.

In addition to `trigger`, the `@consistent` annotation has the following parameters:

1. `period` - specifies how often to establish a consistent state. This is valid only for periodic consistent regions.
2. `drainTimeout` - specifies when to timeout on the establishment of a consistent state. If the establishment times out, the region is assumed to have failed, and a reset attempt is made.

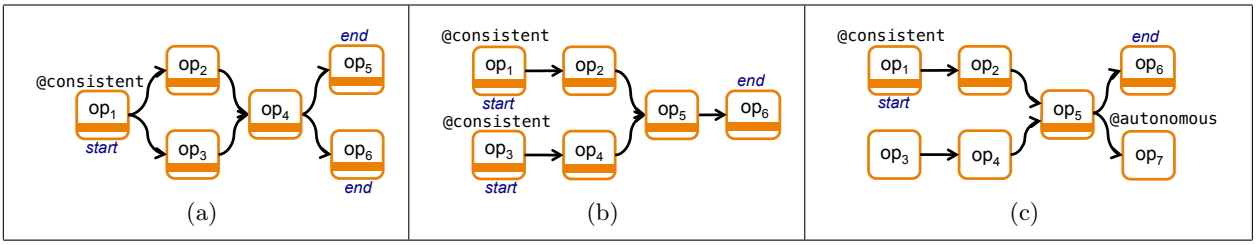

Figure 1: Example definitions of a consistent region. Figure 1(a) shows an application with a single consistent region with a single start operator and two end operators. Figure 1(b) shows a single consistent region with two start operators and one end operator. Figure 1(c) shows a single consistent region and two autonomous regions (Operator op_5 has a consistent and an autonomous input port).

3. `resetTimeout` - specifies when to timeout when resetting the region to a consistent state. If the reset times out, a new reset attempt is made.
4. `maxConsecutiveResetAttempts` - indicates how many consecutive attempts are made to reset the region to a consistent state. If the region still cannot be reset, the region is assumed to have failed and administrator intervention is required to reset the region.

Figure 2 shows the SPL program for Figure 1(a). Except for `trigger` and its conditionally dependent parameter `period`, all other parameters have default values.

```
00: composite Main {
01:   graph
02:     @consistent(trigger=periodic, period=10.0)
03:     stream<int a> Op1 = Beacon() {}
04:     stream<int a>  Op2 = Functor(Op1) {}
05:     stream<int a>  Op3 = Functor(Op1) {}
06:     stream<int a>  Op4 = Functor(Op2, Op3) {}
07:     () as Sink1  Op5 = Custom(Op4) {}
08:     () as Sink2  Op6 = Custom(Op4) {}
09: }
```

Figure 2: Sample SPL program equivalent to the application graph in Figure 1(a). `@consistent` is placed as an annotation to the source operator (`Beacon`).

3.2 Autonomous Annotation

An operator that is not in a consistent region is called *autonomous*. This means that during runtime the output streams and the state of these operators are not coordinated with any other operator in the application. Interconnected autonomous operators are considered to be in an autonomous region. Operators are autonomous by default.

Autonomous source operators do not replay tuples on a failure. When in the upstream of operators in a consistent region, autonomous operators process tuples at-most-once. When in the downstream of a consistent region, autonomous operators may receive duplicate tuples. This is because start operators of a consistent region replay tuples on a failure.

The `@autonomous` annotation works similarly to `@consistent`. Once placed on an operator, all the operators in its reachability graph are considered autonomous. If an operator is in the reachability graph of both an autonomous and a consistent annotation, then the consistent annotation takes precedence, and the operator is considered to be in the consistent region. An operator can consume output streams from both autonomous and consistent operators as long as the streams are connected to different input ports. When this situation occurs, input ports consuming streams from consistent operators are called *consistent input ports* and input ports consuming streams from autonomous operators are called *autonomous input ports*. When a consistent operator has an autonomous input port, it means that the operator can be made consistent independently of that stream.

Figure 1(c) shows an example of an application that uses both `@consistent` and `@autonomous`. When placing the `@consistent` annotation in op_1, operators op_2, op_5, op_6 and op_7 would be included on the consistent region. As op_7 has an autonomous annotation, it starts a new autonomous region. If this operator had downstream operators, they also would be in the autonomous region. Operators op_3 and op_4 have no annotations and are by default autonomous. This results in op_5 having an autonomous input port.

3.3 Code Generation

We leverage the SPL code generation framework to enable the generation of code specific to consistent regions only when an operator is indeed in a consistent region. This is important to reach the goal of having near zero performance impact on applications that do not require guaranteed tuple processing. By using the code generation framework, C++ primitive operators can query a *consistent region context* to discover if the operator is in a consistent region, among other information. One important use case of the consistent region context is to enforce compile-time checks. In this way, developers can enforce that only certain operators and configurations are allowed in a consistent region. The SPL compiler also leverages the context to check if a Custom operator is used in consistent regions and automatically generates serialization and de-serialization code for stateful Custom operators to checkpoint and reset its state.

4. RUNTIME

In this section, we detail the runtime implementation of consistent regions. We first describe the consistent region controller, which is responsible for failure detection and the protocol coordination. We then present the protocol at the operator level, detailing the rules for punctuation propagation and for persisting/restoring operator state. Finally, we describe how to ensure FIFO order in stream connections.

4.1 Consistent Region Controller

The consistent region controller is the central component for coordination. The controller is responsible for notifying the start operators of the region to begin the establishment of a consistent state. It also tracks which operators of the region have completed the drain and checkpoint stages. On completion, the controller notifies start operator(s) that the region can resume tuple processing. The controller is also

responsible for notifying a region that it must reset when PE or controller failures occurs. Each consistent region has a dedicated consistent region controller.

The controllers reside in a special operator called *Job Control Plane* (JCP). The JCP operator is a general purpose component in SPL that enables operators to exchange out-of-band control information. The underlying architecture is the standard Java Management Extensions (JMX) [23]. Operators can create services implemented as management beans (MBeans) in the JCP and interact with them using JMX. The JCP hosts a set of pre-installed services. The consistent region controller subscribes to one such service called *Region Monitor* to get notifications regarding PE health.

The Region Monitor service is the core component for associating application failures to a consistent region and triggering a reset. The Region Monitor is not responsible for detecting failure itself, but it listens to all application health related notifications from various IBM Streams infrastructure services and emits those notifications that are related to the PEs hosting operators in consistent regions. It reports two types of failures that can lead to tuple loss: PE crashes (e.g., process crash or host crash) and PE interconnection failures (e.g., a socket disconnects).

The controller maintains information to track the progress of establishing or restoring of a consistent state. Upon any transition of the protocol (e.g., when the region finishes checkpointing), the controller persists its state to the Zookeeper service in the IBM Streams infrastructure. This allows the controller itself to restart from crash.

The controller is not involved in restarting PEs, relocating PEs on host failures, or mending broken PE interconnections. These actions are performed by the Streams platform. The controller is responsible only for coordinating consistent state establishment and restoration.

4.2 Protocol

The establishment and restoration of consistent states use a token-based protocol, where special punctuations flow through the stream processing graph of a consistent region. In this section, we provide more details on how punctuations are propagated among operators and how cycles in topology and autonomous regions are handled as part of the protocol.

4.2.1 Establishing a Consistent State

The process of establishing a consistent state has two main stages, namely *drain* and *checkpoint*. These stages have a local effect (i.e., in a single operator) and a global effect (i.e., for the whole region).

Drain. Locally, the drain stage enables an operator to submit any pending tuples to its output streams or external systems. During drain stage, the operator is still performing useful work towards its progress, as the operator performs actions that are related to its semantic. For example, an operator that writes tuples to an output file (FileSink), can flush the current output stream it is writing to. On the other hand, a Filter operator, which does all its filtering decision as it receives a tuple, does not need to take any action during the drain stage, as it has no pending tuples. After the drain stage finishes, operators can no longer *submit* new tuples. Globally, the drain stage enforces that the region stops producing new tuples and that in-flight tuples get processed. In-flight tuples are tuples currently being processed by operators (e.g., in the middle of a transformation), tuples in the SPL runtime buffers (e.g., thread queues), and tuples flowing through PE inter-connections (e.g., TCP sockets).

Checkpoint. Locally, operators serialize and persist their state in the checkpoint stage. A checkpoint always occurs *after* a drain. FileSink can checkpoint the file position of the output file stream, whereas a stateless Filter has no data to checkpoint. Globally, a consistent state is achieved when all operators in the region complete the checkpoint stage.

These two stages are triggered as a result of notifications from the consistent region controller and the flow of the special punctuations. The protocol begins at the start operators of the region. In a periodic region, the consistent region controller sends a notification to start operator(s) according to the configured period. Once a start operator receives the notification, it executes the drain stage, stops submitting new tuples, and executes the checkpoint stage. After that, the SPL runtime submits *DrainMarkers* to all output stream connections of the start operator. In an operator-driven region, the process is the same, except that the establishment of the consistent state starts as a result of a direct request from the start operator to the consistent region controller.

For non-start operators, the drain and checkpoint stages occurs only after processing a specific number of markers. The number of expected markers depends on the kinds of input ports of an operator and the number of incoming stream connections to each input port.

There are two kinds of input ports in SPL: *data* ports and *control* ports. When processing tuples from data ports, an operator can change internal state and submit tuples to its output streams. When processing tuples from control ports, an operator can only change internal state. When in a consistent region, an operator input port can also be qualified as *consistent* and *autonomous*. Consistent input ports are ports in which all its stream connections come from operators that are also part of the consistent region. Autonomous input ports are those in which all incoming stream connections come from operators that are autonomous. Data ports and control ports can be either autonomous or consistent.

An operator enters drain stage when it receives a *DrainMarker* in all its input stream connections of all its *consistent data* ports. When a DrainMarker is processed by an operator in a given stream connection, it implies that all tuples *prior* to the DrainMarker were processed. As a result, the stream connection is empty and all the state of that stream connection is reflected as operator internal state and/or new tuples on the output streams of the operator. DrainMarkers from consistent control ports are not required. At this moment, the operator is free to further submit any pending tuples. Once that is complete, no new tuples are submitted downstream. If any tuple arrives from an *autonomous data* or *autonomous control* port, its processing is blocked and is not allowed to proceed until the protocol has finished its execution. If the operator has no consistent control ports, the operator proceeds to the checkpoint stage. If any control port is present, then the DrainMarker is submitted downstream without the completion of the checkpoint stage.

The checkpoint stage starts after the drain stage completes and the operator processes DrainMarkers from consistent control ports. As control ports do not generate further tuple submission, it is guaranteed that processing its tuples only changes operator internal (process) state and not output stream connection (channel) state. This special rule regarding control ports allows our protocol to reach a

consistent state even in regions that have cycles. This is because the drain stage can complete *before* processing markers from connections that might be cyclic, and the checkpoint completes only *after* processing markers from these same connections. This ensures that all tuples flowing through the cycle are processed and reflected as operator state.

Figure 3 shows an example graph with operators op_{1-5} in a consistent region and operator op_6 being autonomous. Operator op_3 has 3 input ports: one consistent data port consuming data from op_1 and op_2, an autonomous port consuming data from op_6, and one consistent control port consuming data from op_4. After processing 2 DrainMarkers from the first port, it drains and submits the marker downstream. As its second input port is autonomous, no DrainMarkers are expected. After op_4 drains and submits the DrainMarker to its output ports, op_3 processes it and starts its checkpoint stage.

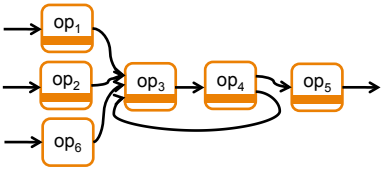

Figure 3: Operator op_3 starts the drain stage after processing markers coming from op_1 and op_2. Its checkpoint stage starts only after processing the marker coming from op_4.

The protocol ends when all operators in the region have completed both stages. The end of the protocol is detected by the consistent region controller. Operators indicate they have completed the protocol by invoking a method in the consistent region controller service. To reduce contention at the consistent region controller, operator notifications back to the controller are batched on a PE basis.

The controller assigns each consistent state a strictly increasing sequence identifier, starting at 1. DrainMarkers carry the sequence identifiers to enable the SPL runtime to group the persisted state of individual operators to a global state. The sequence number of 0 represents the application initial state, prior to the processing of any tuple.

4.2.2 Restoring a Consistent State

Restoring a consistent state has a single stage called *reset*. In this stage, each operator fetches its state from the checkpoint backend store and de-serializes it, replacing the current values of its variables among other actions. The FileSink operator can restore its current position on its output file, seek for that position, and truncate the file. For Filter, no action is needed on a reset. Globally, the collection of states restored by all operators in the region are guaranteed to be consistent.

The protocol to restore a consistent region is similar to the one for establishing a consistent state. A restoration is triggered by the consistent region controller after the detection of a failure. Once start operators receive the notification from the controller, the SPL runtime stops the submission of new tuples. After the operator is paused, the reset stage starts, and a *ResetMarker* punctuation is submitted to all output ports of the operator. The ResetMarker goes through the operator stream connections after all tuples previously submitted by the operator.

For operators that do not start the region, the reset is triggered after receiving one ResetMarker per input stream connection. The condition to trigger a reset is the same as the condition to trigger a checkpoint. While resetting the consistent region, tuples sent prior to ResetMarkers are still processed by operators. Although this design decision may increase the recovery time, it avoids the need to add extra thread synchronization primitives in the main tuple processing path. As a result, we favor faster tuple processing during normal operations over faster recovery times, as recovery is expected to be far less frequent.

The only situation in which the SPL runtime explicitly drops submitted tuples while resetting is when tuples are submitted through an output PE connection that has been broken and reestablished. This happens when the downstream PE crashes or the TCP connection breaks. If such tuples were to go through, downstream operators would process tuples from an incomplete stream or while its state is not yet consistent. This explicit tuple drop takes place within an already synchronized code section, so the only extra cost during normal operation is a conditional statement.

Similar to DrainMarkers, a ResetMarker carries a sequence number that identifies the last successfully established consistent state. It is used by the runtime to fetch the correct operator state from the checkpoint backend store. ResetMarkers also carry a reset attempt which is used to handle failures that occur while the region is being reset.

When a region cannot complete a reset within the timeout specified in the annotation, the consistent region controller attempts a new reset. If resets fail consecutively and reaches the `maxConsecutiveResetAttempts` specified in `@consistent` annotation, the region halts and administrator intervention is required.

4.2.3 Protocol Optimizations

We apply several optimizations to improve protocol performance. The first is to do early marker forwarding and to dispatch operator state serialization and persistence to background threads. When establishing consistent states, the SPL runtime forwards DrainMarkers right after it finishes the drain stage. During restoration, ResetMarkers are forwarded once the stream connections are cleaned up and before starting the reset stage. This enables operators in a serial chain to checkpoint and reset concurrently. The second optimization is for the background threads to batch checkpoint persistence requests of different operators. This reduces the number of I/Os needed to checkpoint multiple operators. The third optimization is offering *non-blocking checkpoint* for operators that implement *user-level copy-on-write*. Such operators are provided with a *prepare-to-checkpoint* stage which is executed after the drain stage. The operator can impelement logic to prepare itself to be checkpointed later (e.g., making a copying of the state). Once this prepare state finishes, DrainMarkers are forwarded downstream and some background thread starts checkpointing the operator (e.g., persisting the copied state). Tuple processing can resume while the background checkpointing is still ongoing. Operators can implement various copy-on-write schemes [12] as needed. Blocking and non-blocking operators can co-exist in the same consistent region. This optimization reduces the blocking time of checkpointing.

4.3 Enforcing FIFO in Stream Connections

As described in Section 4.2.1, our protocol strictly relies on DrainMarkers to ensure that a given set of tuples were

processed by the consistent region. The first key requirement for the correctness of our protocol is that stream connections must be First-in, First-out (FIFO). This applies to both failure-free and failure scenarios. The second key requirement is that DrainMarkers cannot be forwarded by operators when failures occur. This ensures that consistent states are not successfully established when in fact tuple processing has failed and tuples must be replayed.

At the language level, SPL provides a uniform way to express stream connections between operators. During application execution, there are three different implementations for those connections. The first two kinds are used when operators are in the same PE. In such cases, a tuple submission maps to a function call or to a tuple copy in a queue. Such queue is then consumed by a different thread to increase pipeline parallelism. The third kind of connection is TCP connections between operators residing in different PEs.

For intra-PE connections, FIFO is naturally established as the SPL runtime fully finishes the execution of a function or the enqueuing of a tuple in a synchronized queue before acting on the next tuple [3]. In these cases, a marker is guaranteed to be consumed after the tuples that submitted prior to it. A failed execution occurs when an operator throws an exception (e.g., operator cannot access a database). The SPL runtime catches the exception and ensures that no other tuple or DrainMarker is processed. The exception is re-thrown and the PE is gracefully shutdown and restarted.

For inter-PE connections, we can rely on TCP/IP to guarantee FIFO in failure-free scenarios. In failure scenarios, we have different TCP connections for the same logical connection. This means we must enforce FIFO across TCP connections: the old connection (before a failure) and the new connection (after a reconnection). If no action is taken, the receiver operator may end up consuming tuples from both connections at the same time, up until the error is detected in the old connection. To solve this problem, our inter-PE connections have a special handshake that checks the incarnation of a given logical connection. Once a new connection request is established, any pre-existing connection from that same logical connection is closed, and only tuples from the new incarnation of the connection are consumed.

As described in Section 4.2.2, the SPL runtime proactively drops tuples (and DrainMarkers) when it detects that a PE connection breaks. This ensures that no new consistent state is established if there was the possibility of tuple loss. If a failure happens right after a DrainMarker submission, the establishment of the consistent state can still complete. Allowing this situation is correct, as TCP guarantees FIFO within a single connection. As a result, if the downstream operator consumes the DrainMarker, it is because it has consumed all tuples before it.

5. OPERATOR PROGRAMMING MODEL

Most of the time, SPL developers compose an application by using pre-existing operators. When new logic is required, developers can write new operators by implementing callback handlers which are invoked when a new tuple is available for processing in one of the operator's input ports. The handler applies a transformation on the input tuple (e.g., filtering, modification of internal state) and optionally generate a new output tuple as a result. The output tuple is submitted to one of the operator's output ports.

Developers can also write multi-threaded operators. In this case, the SPL runtime creates as many threads as specified by the operator and lets each thread execute a user-defined callback, giving full control of the thread to the operator. This is commonly used by source operators to control data ingestion and tuple submission. Non-source operators can also be multi-threaded. This can be for performance reasons or to execute tasks asynchronously to tuple processing (e.g., a time-based window).

Operators can be written in C++, Java, and SPL itself. When using C++, developers can leverage a code generation framework to generate specialized code depending on the operator configuration. Java and SPL operators (also called Customs) must use runtime APIs to customize operator behavior. To fully leverage the power of consistent regions, developers must adapt operators that are stateful or that interact with an external system (e.g., database, file system). To enable operators for consistent regions, we have added a new interface named `StateHandler`. The interface exposes directly to operators the different stages of the protocol, namely *drain*, *checkpoint*, and *reset*. This empowers operator developers to clearly define how a given operator behaves when in a consistent region, and even achieve exactly-once application output in certain situations.

Figure 4 shows an example of a C++ operator that counts the number of processed tuples and sends the result downstream. As this operator does all its transformations and tuple submissions when processing the tuple (lines 02-09), no drain logic is needed (line 11). At the checkpoint stage, the operator serializes and persists its state to the checkpoint backend store (lines 13-16). At the reset stage, the operator deserializes the state from the checkpoint backend (lines 18-21). The `resetToInitialState` callback is a special case of the reset stage, so that an operator can reset to its initial state after the execution of the constructor. This is used when there is an operator failure prior to the successful establishment of the first consistent checkpoint. For all `StateHandler` methods, developers must use a lock guard (`AutoMutex`) when accessing state, as the callbacks might be invoked by a runtime thread.

Figure 5 shows an operator coded in SPL with the same functionality as the one in Figure 4. This operator produces an output stream named TupleCounter with two attributes (line 0). When a tuple comes in through the Input stream (line 1), it increments the counter and submit a new tuple to the output stream (lines 4-8). When a Custom operator is in a consistent region, the SPL compiler generates extra code to serialize and deserialize operator state (line 3). As a result, developers do not need to take any special action and can avoid the boilerplate code of C++ operators.

5.1 Multi-Threaded Operators

A key requirement to achieve consistency is for the checkpointed operator state to be coordinated with the state of the stream connections. If the operator changes its internal state and submits a tuple as a result, we must ensure that if we save the operator state after the change, then the tuple must indeed be submitted to downstream operators. This means that there must be a point in the operator code in which it gives the SPL runtime the opportunity to do state

[3]This behavior is slightly different for multi-threaded operators. Details on how such cases are addressed can be found in Section 5.1.

```
00: MY_OPERATOR::MY_OPERATOR() : numTuples_(0) { }
01:
02: void MY_OPERATOR::process(Tuple const & tuple,
03:   uint32_t port) {
04:   AutoMutex am(mutex_);
05:   numTuples_++;
06:   OPort0Type otuple(tuple.getAttributeValue(0),
07:     numTuples_);
08:   submit(otuple, 0);
09: }
10:
11: void MY_OPERATOR::drain() { }
12:
13: void MY_OPERATOR::checkpoint(Checkpoint & ckpt) {
14:   AutoMutex am(mutex_);
15:   ckpt << numTuples_;
16: }
17:
18: void MY_OPERATOR::reset(Checkpoint & ckpt) {
19:   AutoMutex am(mutex_);
20:   ckpt >> numTuples_;
21: }
22:
23: void MY_OPERATOR::resetToInitialState() {
24:   AutoMutex am(mutex_);
25:   numTuples_ = 0;
26: }
```

Figure 4: Operator implemented in C++ can specify its behavior in consistent regions by implementing the *drain*, *checkpoint*, *reset*, and *resetToInitialState* callbacks.

```
00: stream<int32 id, int32 counter> TupleCounter =
01:                                   Custom(Input) {
02:   logic
03:     state: { mutable int32 count = 0; }
04:     onTuple Input: {
05:       count++;
06:       submit({id = Input.id, counter = count},
07:              TupleCounter);
08:     }
09: }
```

Figure 5: Operator implemented in SPL. Implementation of consistent region callbacks are automatically generated by the compiler.

serialization and the drain marker propagation.

In operators that do processing only as a result of an incoming tuple (Figures 4 and 5), a natural point in which the runtime can do the checkpoint is right after fully processing the tuple (`process()` and `onTuple`). After the method executes, the operator has done internal state change (if any) and optionally submitted tuples downstream.

When the operator has full control of the submission thread, defining the correct point becomes problematic. For example, a source operator commonly has a loop that consumes data from an external source, creates a tuple and submits it downstream. The only opportunity that the SPL runtime has to checkpoint the operator state is when the operator code interacts with the runtime, such as submitting a tuple. However, checkpointing operator state right before or right after submitting a tuple downstream is not necessarily correct. This is because the developer is free to write any code before and after the submission, including doing multiple tuple submissions. As a result, the runtime cannot correctly infer the semantically correct point to stop the operator and checkpoint its state.

To solve this problem, we introduce the concept of *consistent region permits*, which enable an operator to bundle state changes and tuple submissions in a single *transaction*. Permits are essentially a semaphore with extra logic for handling the stages of the consistent region protocol. With them, developers can ensure that the serialized internal state correctly reflects the state of the channels. To start a bundled state change and tuple submission, operator code must acquire a permit. Once the operator successfully acquires a permit, it can do state changes and submit resulting tuples to the operator's output ports. It is illegal for an operator to submit a tuple without holding a permit.

While the operator is holding permits, the SPL runtime cannot start the establishment or the reset of a consistent state. When the SPL runtime receives a notification from the controller that it must establish a consistent state, it will indicate that the operator must drain. While draining, the operator can continue to submit new tuples. Once the drain callback returns, the runtime no longer gives out new permits and waits until all existing permits are released. When attempting to acquire a new permit, the operator blocks until it is safe to resume tuple processing. When the operator is not holding any permits, the SPL runtime can checkpoint and reset state. When the permit acquisition is granted to an operator thread after a checkpoint, the thread continues its normal operation and can submit the next tuple. When the permit is granted after a reset, the operator might have changed its state to an older state. The next tuple it will submit will be using the recovered state.

Figure 6 shows a code segment of a source operator that submits tuples with strictly increasing values. This thread only exits when the job is about to shutdown (line 2). At every loop iteration, the operator acquires a permit, creates and submits a tuple, and update its internal state. The permit bundles the tuple submission and the state change (lines 6-7). The permit is acquired at every loop iteration. This gives the SPL runtime an opportunity to do a checkpoint or a reset at every tuple. Permit acquisitions involve locking, which might be costly for some operators. This can be easily addressed by doing multiple tuple submissions in a single permit acquisition. The code for checkpointing and resetting the state of this operator is identical to Figure 4.

```
00: void MY_OPERATOR::process(int32_t threadId) {
01:   ProcessingElement & pe = getPE();
02:   while(!pe.getShutdownRequested()) {
03:     ConsistentRegionPermit crp(_crContext);
04:     AutoMutex am(mutex_);
05:     OPort0Type tuple(numTuples_);
06:     submit(tuple, 0);
07:     numTuples_++;
08:   }
09: }
```

Figure 6: Source operator doing a consistent region permit acquisition to bundle tuple submission and state change on a free running thread.

In general, permits can be used by any operator that is multithreaded and do tuple submission from the threads (e.g., a time-based aggregator). Permits are also used by the SPL runtime when an operator in the consistent region has an autonomous input port (Figure 1(c)). During compilation, the SPL compiler automatically generates permit acquisition code for the autonomous input port. In this way, we can guarantee that no tuples flow through the input ports while the region is checkpointing and resetting.

5.2 Operator Adaptation

SPL allows developers to reuse and share analytics via toolkits. A toolkit contains pre-built analytics which can be used by any SPL application. SPL has a standard toolkit, which has general-purpose operators (aggregation, join, filtering, data ingestion, parsing, and data exporting), and several specialized toolkits, which include special purpose operators (e.g., time-series analytics). Currently, 73 operators in the standard and specialized toolkits support consistent regions. With respect to consistent regions, toolkit operators can be split into two categories: (i) operators with in-memory state only, and (ii) operators with external state.

5.2.1 Operators with In-Memory State

Operators with in-memory state can be further divided in three categories: (i) operators with state in serializable variables, (ii) operators with part of its state in libraries, and (iii) operators with blocking calls when handling tuples.

Operators in category (i) are trivial to adapt. Developers just need to serialize and deserialize its member variables. Examples are our operators that do aggregation and join.

Operators in category (ii) require libraries to have support for serialization and deserialization. If they don't, we limit the conditions which we do checkpointing for. One example is the operator for stream decompression (Decompress), which uses the Boost libraries for decompression. Establishing consistent states is allowed only when Decompress finishes decompressing a complete stream, which is when the library holds no state. This condition can be easily achieved with consistent regions, as we enable operators to define the boundaries in which consistent states are established.

Operators in category (iii) are those that block while processing a tuple, and are unblocked only by processing another tuple through a different thread. One example is the Gate operator that only submits a new tuple once it receives an acknowledgment tuple corresponding to the previously submitted tuple. Under failures, there is no guarantee that such tuple will come and cause the main processing thread to unblock. As a result, the `process()` method will not finish, and the condition to establish a consistent checkpoint will not be met. The standard toolkit has 2 such operators and they are are not supported in consistent regions.

5.2.2 Operators with External State

Operators with external state are any operators that interact with an external system. In general, they are source and sink operators, which are used to ingest data into the application or to externalize the application result. They can also be intermediate operators that interact with, for example, a database or a key-value store to do lookups.

For input adapter operators, only operators that ingest data from a replayable streams were adapted for consistent regions. Examples include a directory scanner, a file source, and a database reader. Operators from non-replayable streams, like a TCP socket reader and a UDP socket reader, do not support consistent regions. Alternatively, we provide an operator named ReplayableStart that buffers input streams and replays them, if necessary, during a consistent region reset.

For operators that write data to an external system, only those that can control the state of the external system were adapted to consistent regions. For example, operators that write tuples to TCP or UDP sockets cannot retract their write. On a failure and data replay, these operators end up writing duplicate data. Operators interacting with external systems that expose sufficient interfaces for retracting writes were adapted for consistent regions and, in many use cases, achieve exactly-once tuple processing semantic. Examples include the operator to write tuples to files (FileSink) and the database writer operator (ODBCAppend).

6. CHECKPOINT BACKEND STORE

Operators' checkpoints are stored in a remote persistent key-value store. An operator writes multiple checkpoints throughout its life cycle. Each checkpoint consists of serialized variables. We organize checkpoints in a key-value store through a two-level data model mapping.

We first represent the checkpoint data in an abstract key-value store data model, and then map this abstract model to the underlying data store's specific data model and data structures. In this abstract model, a key-value store provides a two-level hierarchical namespace. There are one or more *Data Store Entries* in the store. A Data Store Entry has a unique name and contains a collection of key-value pairs. For each key-value pair, the key is a byte string and is unique within the scope of the Data Store Entry; the value is also a byte string whose size is no larger than a configured size limit. The model requires that the key-value store provides interfaces to (1) create, delete, and test existence of a Data Store Entry; (2) put, get, delete, and test existence of a key-value pair within a Data Store Entry. This abstract key-value store data model is general and can be implemented on top of popular key-value stores. For example, the Data Store Entry abstraction can be implemented as a hash table in Redis [8] or a database instance in LevelDB [7] which holds all key-value pairs in the Data Store Entry.

Each operator stores its checkpoints in a Data Store Entry that is uniquely named by a concatenation of application's job ID and operator's index within the application. Both application job ID and operator index are assigned by the system. Within an operator's Data Store Entry, each checkpoint is stored as a number of key-value pairs. The serialized checkpoint data are broken into fix-sized chunks (except the last chunk which may be smaller), and each chunk is assigned a key that is a concatenation of checkpoint sequence ID plus a chunk index. The chunk size is set to accommodate the underlying key-value store's value size limit.

We provide both C++ and Java checkpointing API for operator developers. As described earlier, an operator in a consistent region should implement the `checkpoint()` and `reset()` callbacks. Both callbacks take a `Checkpoint` instance as parameter. The `Checkpoint` class provides an interface to write data from and read data to the checkpoint backend store (see Figure 4 for example).

The SPL runtime implements the abstract key-value store model in the form of client adapters to specific key-value stores, and handles serialization, chunking, I/O batching, data transfer, sharding and replication under the hood of the checkpointing API. We currently support using Redis and LevelDB as a checkpoint backend store.

7. EXPERIMENTAL EVALUATION

In this section, we describe tests that are continually conducted for validating the functionality of consistent regions. We then present performance experiments to show

the impact of consistent regions on application performance. Those performance experiments are conducted with 4 machines. Each machine has a Intel Xeon E5-2680 processor with 16 cores at 2.70GHz, 64KB L1 cache, 256KB L2 cache, and 20MB shared L3 cache, and 250GB DRAM, and connected with 1Gigabit Ethernet. We use Redis as the checkpoint backend store. Redis version 2.8.9 is used with snapshotting and write-ahead-logging enabled.

7.1 Implementation Validation

The first question we investigate is, *Do consistent regions provide guaranteed tuple processing?* For that, we have two sets of tests. The first set validates whether the protocol can correctly establish and restore consistent states in topologies composed only of simple operators (e.g., filtering, tuple counting). The second set is to validate each adapted toolkit operator (e.g., aggregate, join) and make sure it can correctly reset its state from a checkpoint upon failure.

The first test set covers a variety of topologies, including those with (i) cycles, (ii) multiple source operators, (iii) multiple sink operators, (iv) multiple consistent regions, and (v) operators with consistent and autonomous input ports, among other variations. This set also exercises a variety of failure modes, such as (i) operator crashes during normal data processing, checkpointing, and reset, (ii) controller crashes during its different state transitions, (iii) crashes of different IBM Streams infrastructure components, including the checkpoint backend, (iv) operator crashes due to exception throwing, and (iv) operator crashes concurrently to crash of infrastructure components. For a test to be considered successful, the output must always be complete and the values of produced tuples must be the exact same as if the application had not fail (*golden* run).

The second test set validates the correct recovery of an application when using a given operator and all its different configurations. Taking as an example the Aggregate operator in the SPL standard toolkit, it can be configured with many different windowing options (e.g., sliding, tumbling, punctuation-based tuple eviction). All must be validated under failure conditions and checked against the golden run output. This process is repeated for all operators that support consistent regions (73 operators in version 4.1).

These two sets total over 200 tests that run as part of the continuous IBM Streams build. For every release, both automated tests and additional QA tests must pass.

7.2 Impact on Application Throughput

Impact on stateful applications. The second question we investigate is, *What is the impact of consistent regions on the throughput of an application with stateful operators?* In general, such impact depends on the frequency of establishing consistent states and the total size of checkpointed state. We run two sets of tests to quantify it.

The first set of tests run five applications and vary the frequency of establishing consistent states. The applications are: (i) Enron-DS, (ii) Enron-FS, (iii) LogWatch, (iv) Vwap, and (v) Lois. Both Enron-DS and Enron-FS process the Enron email dataset [6] and do word counting, but they ingest data differently. Enron-DS uses a directory scanner to scan file names for file data ingestion. Enron-FS uses 5 different file sources to read files in parallel. LogWatch detects security attacks by analyzing the system messages from a Linux host. Vwap calculates the volume weighted average price of stocks on incoming trades and quotes feeds. Lois is a radioastronomy application, analyzing radio signals. For each application we measure its performance without consistent region to establish a baseline, and then measure performance with the whole application in a consistent region. Table 1 shows more detailed information about the applications.

Figure 7(a) shows the impact on throughput when using consistent regions on the applications as described in Table 1. For Enron-DS, we use an operator-driven consistent region, where a consistent state is established after fully processing a file. For all others, we use periodic consistent regions, and configure the periods to vary from 2, 4, 8, 32, to 64 seconds. We run each configuration 10 times. The throughput is normalized to the average throughput of the application running without consistent regions.

As expected, the impact on throughput decreases as the frequency of establishing consistent states decreases. The impact goes from 27% on average for LogWatch, to 0.01% on Enron-DS. The impact on Enron-DS is negligible, as the application only establishes consistent states after fully processing a file. Although it has a larger state than the other applications, it only persists it infrequently. Furthermore, when it does so, it only takes 0.18 seconds. When Enron-FS establishes consistent states at every 2 seconds, the application throughput decreases by 14%. In this case, the total elapsed time for taking a consistent state is 1.03 seconds. During this time, the application is still processing data in the drain stage, but no *new* data is being pushed down the pipeline. As the period of the consistent region increases, the impact on the average application throughput becomes negligible. For Vwap, the state is small (2.3MB), and the drain and checkpoint stages take only 0.07 seconds on average. Hence the impact on Vwap throughput is small (9.1% to 4.7%). The impact on the Lois application is interesting. Even with a period of 2 seconds, the impact on throughput is only 4%. This is because the computation cost per tuple is high, so most of the time spent during a consistent state is in the drain stage, when the application is still processing tuples. Once the consistent state is established, the application queues are empty. However, the impact of an empty queue is diminished because once there is any tuple in the queue, that tuple takes a long time to process.

The second set of tests run a synthetic application and vary the total size of checkpointed state. The synthetic application has one source operator sending tuples to a chain of 64 downstream operators. One of the downstream operators maintains a sliding window for incoming tuples. We implement two versions of this operator: (i) the *blocking checkpoint* version persists the sliding window in checkpoint stage; whereas (ii) the *non-blocking checkpoint* version makes a copy of the window in prepare-to-checkpoint stage and persists the copy in the background after tuple processing resumes (as described in Section 4.2.3). We run the application without consistent regions to form a baseline. We then set the whole application in a consistent region and run the blocking and non-blocking checkpoint versions, respectively. We vary the sliding window size but fix the consistent region period to 8 seconds. Figure 7(b) shows the throughputs with blocking and non-blocking checkpoint, both normalized to the same baseline throughput.

With blocking checkpoint, the throughput degrades more severely with larger checkpoint sizes (from 4% with 8MB checkpoint to 40% with 512MB). This is because the tuple

Application	Number of operators	Number of PEs	Max. fan-out	Max. graph length	Average global state size (MB)	Average baseline throughput (tuples/sec)	Average time of a drain (sec)
Enron-DS	20	3	5	6	12.00 (+-0.00)	11781.82 (+- 244.85)	0.18 (+- 0.04)
Enron-FS	22	3	5	5	12.06 (+-0.07)	25445.40 (+- 188.82)	1.03 (+-0.46)
LogWatch	29	7	7	13	7.1 (+-3.6)	192494.56 (+- 9717.08)	1.16 (+-0.28)
Vwap	26	6	4	6	2.37 (+-0.14)	384245.91 (+- 4729.49)	0.07 (+-0.01)
Lois	22	1	3	16	1.4 (+-0.0)	1855.71 (+- 18.18)	1.14 (+-0.096)

Table 1: Application characteristics in terms of number of operators, number of PEs, maximum fan-out and graph length in toplogy, average size of checkpointed global state, baseline throughput, and average time to establish a consistent state. The last two measurements include the 95% confidence interval.

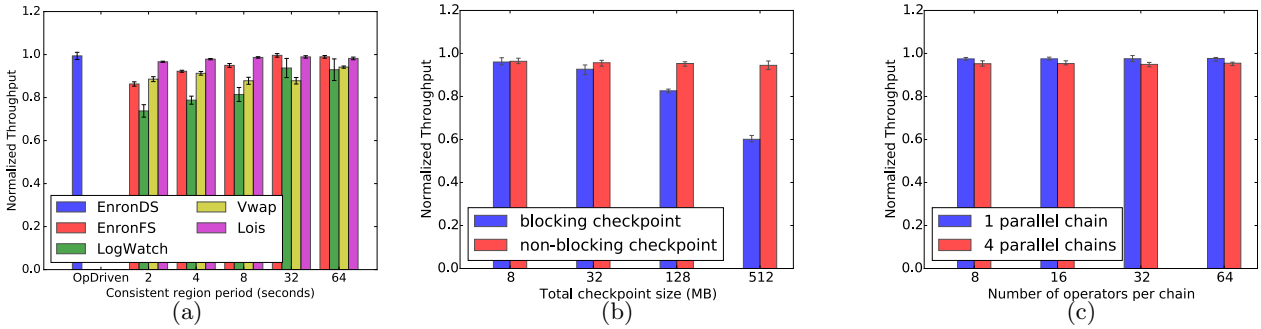

Figure 7: Effect of consistent regions on application throughput. Figure 7(a) shows the normalized throughput on stateful applications when varying the period. Figure 7(b) shows the normalized throughput when varying checkpoint sizes. Figure 7(c) shows the normalized throughput when scaling a stateless application topology.

processing is paused until the checkpoint stage completes (i.e., until checkpoint is written to backend store). On the other hand, non-blocking checkpoint resumes tuple processing shortly after the prepare-to-checkpoint stage finishes. In this test case, the prepare-to-checkpoint stage makes a copy of the sliding window, which takes much less time than persisting the data to backend store. As a result, non-blocking checkpoint can sustain high normalized throughput even with large checkpoints (e.g., 94% with 512MB checkpoint).

Impact on stateless applications. The third question we investigate is, *What is the impact of consistent regions on the throughput of an application with stateless operators only?* This question is of interest because the impact reflects the overhead of marker propagation. To answer this, we use a synthetic application consisting of a source operator, a sink operator, and multiple stateless operators in between. These stateless operators are organized as a number of parallel chains. The source operator sends tuples to those parallel chains in a round-robin manner. All the parallel chains are connected to the sink oprator which drops all tuples it receives. This synthetic application emulates the data parallel and pipeline parallel patterns. We vary the number of operators in a parallel chain from 8, 16, 32, to 64. Every 8 operators are fused in a single PE. The source and sink operators are in two other PEs. Adjacent PEs are placed on different machines. We also vary the number of parallel chains to be 1 or 4. We measure throughput when the application is in a consistent region with a 8-second period, and normalize the throughput to the baseline in which the application runs without consistent regions.

As shown in Figure 7(c), when the topology has one parallel chain, the impact on throughput of stateless applications is at most 3%, with little variation among the different chain lengths. The drain times vary from 38ms, for 8 operators, to 67ms for 64 operators. As expected, the drain time increases as the topology increases. Still, the throughput does not show much variation among the different chain lengths, as there is still tuple processing occurring concurrently with tuple drain. Similar results are observed when there are 4 parallel chains, although the impact on throughput is slightly larger (around 4.6%) than the single chain case. This is because the tuple flow is blocked until the sink operator receives drain markers from all parallel chains. More parallel chains lead to longer blocking times.

8. RELATED WORK

Many of the earlier works in stream processing considered low-latency to be a key requirement. This was one of the primary drivers for approaches such as the one in Borealis [10], where tuples with tentative results were generated when an application would experience failures and later corrected after recovery. Other techniques supported partial fault-tolerance [11, 17, 19, 20, 22, 27] and sacrificed application output precision for lower runtime costs. In [19], we proposed using language-level abstractions to annotate which operators in a stream processing graph do checkpointing. With this approach, operators checkpoint independently, so no tuple processing guarantee is provided.

Apache Storm [3] by default provides operator restarts without data processing guarantees. Storm has a tuple acknowledging scheme for processing tuples at-least-once [4]. This scheme enables developers to explicitly describe the provenance of a tuple. This acknowledgment scheme does not support applications requiring in-order tuple processing. Storm's higher level Trident API [5] enables exactly-once tuple processing. With this API, the application can associate operator state to a batch of processed tuples. A batch in Trident is equivalent to a set of tuples processed between two consistent states. In IBM Streams, users are free to define batch sizes according to a period or to be operator-specific (operator-driven). Furthermore, we allow any user-defined operator to be used in consistent regions, including operators that use external systems. One such example is the RScript operator [1], which can use stateful R scripts to do

tuple transformations and participate in a consistent region by implementing the drain, checkpoint and reset callbacks. Heron [21], Twitter's reimplementation of Storm, does not support exactly-once tuple processing.

Spark [25] provides fault tolerance via immutable RDDs and lineage tracking, but limits the applications to deterministic and rollbackable operators. In comparison, IBM Streams allows more general application logic. Furthermore, we provide API that enables operators to take specific actions during different stages of the protocol. This allows applications to interact with external systems and integrate complex behavior into the consistent region protocol.

Meteor shower [24] also uses Chandy-Lamport for establishing consistent states of a distributed streaming application. Unprocessed tuples are persisted together with operator state. Our implementation propagate tokens from sources and does a drain stage to ensure that any pending tuples are fully processed. This has an advantage when tuples contain data that reference external resources and such resources must be periodically recycled. For example, a tuple can reference a file name which can be opened and read by an operator downstream. If the protocol guarantees that a given tuple has been fully processed, it means that the file can be deleted when a checkpoint retires, as there is the guarantee that it has been fully processed.

MillWheel [9] has both exactly-once and at-least-once processing guarantees, similar to IBM Streams. MillWheel, however, takes a different approach, as all its processed records have identifiers which are used in a deduplication step. Such technique is unsuitable for Streams, as records can't always be uniquely identified deterministically.

Apache Flink [2] also uses a variation of Chandy-Lamport to support guaranteed processing [13], similar to what is available since IBM Streams 4.0 [1, 18]. Some key differences in Streams are that we provide programming language abstractions which enable developers to do fine-grained selection of which parts of the topology require guaranteed data processing, lowering the cost of providing fault tolerance. We also provide APIs to support multi-threaded operators to participate in a consistent state.

The sweeping checkpointing technique [15] persists both internal operator state and tuples in output queues. As described above, our method does not persist queue state. Similar to sweeping checkpointing, our method can tolerate multiple operator failures, as tuples are fully processed and checkpoint state is stored in a persistent backend store.

9. CONCLUSIONS

In this paper, we describe how we achieve at-least-once and exactly-once tuple processing in IBM Streams by applying a variation of the classic Chandy-Lamport distributed snapshot algorithm. Such endeavor was non-trivial, as IBM Streams applications can be partially fault-tolerant, be nondeterministic, have cycles in topology, have multi-threaded operators, have several legacy operators, and can directly access external components (e.g., files, databases). Consistent regions have been available in IBM Streams since version 4.0. Since version 4.1, we have implemented incremental checkpointing for windowed operators [26]. In the future, we plan to release the support for non-blocking checkpointing and evaluate the impact of persisting stream connections versus fully draining them.

Acknowledgments

We thank Howard Nasgaard and Ankit Pasricha for the help throughout the development of consistent regions.

10. REFERENCES

[1] IBM InfoSphere Streams Version 4.0. `https://www-01.ibm.com/support/knowledgecenter/SSCRJU_4.0.0/`, March 2015.

[2] Apache Flink. `http://flink.apache.org`, 2016.

[3] Apache Storm. `http://storm.apache.org`, 2016.

[4] Apache Storm. Guaranteeing Message Processing. `https://storm.apache.org/documentation/Guaranteeing-message-processing.html`, 2016.

[5] Apache Storm. Trident State. `https://storm.apache.org/documentation/Trident-state.html`, 2016.

[6] Enron Email Dataset. `http://www.cs.cmu.edu/~./enron/`, 2016.

[7] Google's Leveldb. `https://github.com/google/leveldb`, 2016.

[8] Redis Key-Value Store. `http://redis.io/`, 2016.

[9] T. Akidau et al. Millwheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.

[10] M. Balazinska et al. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1):3:1–3:44, Mar. 2008.

[11] N. Bansal et al. Towards optimal resource allocation in partial-fault tolerant applications. In *INFOCOM*, 2008.

[12] T. Cao et al. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD*, 2011.

[13] P. Carbone et al. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015.

[14] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[15] Y. Gu et al. An empirical study of high availability in stream processing systems. In *Middleware*, 2009.

[16] M. Hirzel et al. IBM streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7, 2013.

[17] J.-H. Hwang et al. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.

[18] G. Jacques-Silva. Guaranteed tuple processing in InfoSphere Streams v4 with consistent regions. `https://developer.ibm.com/streamsdev/2015/02/20/processing-tuples-least-infosphere-streams-consistent-regions/`, February 2015.

[19] G. Jacques-Silva et al. Language level checkpointing support for stream processing applications. In *DSN*, 2009.

[20] G. Jacques-Silva et al. Fault injection-based assessment of partial fault tolerance in stream processing applications. In *DEBS*, 2011.

[21] S. Kulkarni et al. Twitter Heron: Stream processing at scale. In *SIGMOD*, 2015.

[22] L. Neumeyer et al. S4: Distributed stream computing platform. In *ICDMW*, 2010.

[23] Oracle. Java Management Extensions. `http://docs.oracle.com/javase/8/docs/technotes/guides/jmx/index.html`, 2015.

[24] H. Wang et al. Meteor shower: A reliable stream processing system for commodity data centers. In *IPDPS*, 2012.

[25] M. Zaharia et al. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[26] F. Zheng et al. Adaptive incremental checkpointing for high-performance data streaming applications. In *Under submission*.

[27] Q. Zhu et al. Supporting fault-tolerance in streaming grid applications. In *IPDPS*, 2008.