CEC: Continuous Eventual Checkpointing for Data Stream Processing Operators

Zoe Sebepou and Kostas Magoutis
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Heraklion GR-70013, Crete, Greece
e-mail: {sebepou,magoutis}@ics.forth.gr

Abstract— **The checkpoint roll-backward methodology is the underlying technology of several fault-tolerance solutions for continuous stream processing systems today, implemented either using the memories of replica nodes or a distributed file system. In this scheme the recovering node loads its most recent checkpoint and requests log replay to reach a consistent pre-failure state. Challenges with that technique include its complexity (typically implemented via copy-on-write), the associated overhead (exception handling under state updates), and limits to the frequency of checkpointing. The latter limit affects the amount of information that needs to be replayed leading to long recovery times. In this work we introduce *continuous eventual checkpointing* (CEC), a novel mechanism to provide fault-tolerance guarantees by taking continuous incremental state checkpoints with minimal pausing of operator processing. We achieve this by separating operator state into independent parts and producing frequent independent partial checkpoints of them. Our results show that our method can achieve low overhead fault-tolerance with adjustable checkpoint intensity, trading off recovery time with performance.**

Keywords; Continuous Stream Processing; Fault-Tolerance;

I. INTRODUCTION

Sources of continuously-flowing information today are growing in both number and data rates produced [1][9]. Consider for example the streams of call-detail records (CDRs) produced by mobile telephony base stations; credit-card transaction authorizations; stock-price feeds from financial markets; and camera video streams used for surveillance. The need for online processing of such information streams has led to the design of complex event processing systems supporting continuous queries expressed in stream-oriented query languages [4][5]. The data operated on (often referred to as tuples) are associated with a monotonically increasing timestamp, forming a time series. A continuous stream processing query is composed of one or more interconnected operators, each computing a function on incoming tuples. Operators that accumulate state by computing a function over sets (also known as windows) of tuples are known as stateful operators.

An important concern, especially in the case of stateful operators is that a failure in the underlying infrastructure may lead to long recovery times and/or irrecoverable loss of operator state. These risks are not acceptable in many application domains with stringent response time and data reliability requirements. Several fault tolerance solutions have been proposed in the past [6][10][11][19] to address this challenge most of them relying on replicating operator state either in the memories of different nodes or on disk (Section VI provides an overview of these solutions).

A prominent fault-tolerance solution used today is the checkpointing and roll-backward methodology [10], hereafter referred to as CRB. In CRB the recovering node loads the more recent checkpoint and replays events that postdate the checkpoint to reach the pre-failure state. Efficient implementations of CRB use copy-on-write [10][17] mechanisms to reduce the amount of time an operator freezes processing while saving the checkpoint to stable store. Challenges with CRB include its complexity (such as in the implementation of copy-on-write), the associated overhead (exception handling during state updates), startup overhead (one needs to compute what needs to be checkpointed, especially if incremental checkpoints are being used) and limits to the frequency of checkpointing (one cannot start a new checkpoint while the previous one is still in progress). The latter limit affects the amount of information that needs to be replayed during recovery leading to potentially long unavailability periods.

Straightforward applications of CRB to stream processing systems [17] consider the entire operator state when constructing checkpoints (namely all windows that have changed since the previous checkpoint). As such, operators accumulating large amounts of state (that is, several GBs in today's systems) require inter-checkpoint intervals in the order of tens of seconds (for example, a 4GB state can take at least 40 seconds to checkpoint over a 1Gbit/s line, assuming stable storage can sustain the same rate). Incremental checkpoints do not seem to help much in that aspect when state changes between checkpoints are spread across the majority of operator's windows.

One way to improve CRB in stream processing systems is to take advantage of the observation that certain window state transitions (such as the closure of a window) are leaving a "footprint" in the operator's output stream (an output tuple). This "footprint" can also serve as a recovery point for that window in case of operator failure. Intuitively speaking, it contains the information that the state of the window was closed as of the time the output tuple was produced. We can build on this observation by creating other types of window state "footprints" by inserting special output tuples in the operator's output stream, such as when a window opens or the state at an arbitrary point in time. A recovery mechanism can examine the operator's output stream to find the most

recent "footprints" of all windows that comprised the operator state at some point in time. In this paper we formalize the above observations into a novel checkpointing mechanism that we term *continuous eventual checkpointing* (CEC).

CEC splits operator state into parts that can be checkpointed independently and in an as-needed basis. In contrast to conventional approaches where a checkpoint is statically constructed at checkpoint-capture time, in CEC the checkpoint is an evolving entity that is continuously and incrementally updated by adding partial checkpoints to it. Partial checkpoints take the form of control tuples containing window state and intermixing with actual data-carrying tuples (preserving time order) in the operator's output stream. To ensure that the checkpoint is recoverable in the event of a node failure, CEC requires that the output stream be written to stable storage. The interleaving between control and regular output tuples means that the checkpoint is spread over the persistent representation of the operator's output stream, rather than being a single cohesive entity at a single location in stable storage. As such, the CEC checkpoint is not immediately available for recovery but must be reconstructed through a process described in detail in this paper.

Since checkpoints can be performed asynchronously at the pace of the operator's choosing, the operator can exploit a tradeoff between overhead and recovery speed (*i.e.*, how often to write control tuples to stable storage *vs.* how far back to seek into the output queue to reconstruct the CEC checkpoint *vs*. how far back to replay from upstream output queues). By mixing control and regular output tuples in the output stream, CEC can leverage an existing persistence mechanism (originally developed for persisting the operator's output queue [20]) to also persist its evolving checkpoint. Besides being straightforward in its implementation, such a scheme enjoys I/O efficiencies due to the sequential access pattern involved. These characteristics make CEC an efficient CRB methodology for streaming operators that can provide rapid recovery with adjustable overhead characteristics.

Our contributions in this paper include:

- A novel state checkpointing technique for window-based streaming operators that takes independent checkpoints of parts of the operator state in the form of control tuples integrated with regular tuples at the operator's output queue.
- Implementation of the technique (including both failure-free and recovery paths) in the context of the Borealis [8] open-source streaming middleware.
- Evaluation of the prototype system under a variety of scenarios.

The remainder of paper is structured as follows. In Section II we describe the detailed design of the CEC mechanism and in Section III our prototype implementation. In Section IV we present the experimental evaluation of CEC and in Section VI we present related work on fault-tolerant streaming. Finally we draw our conclusions in Section VII.

II. Design

In this paper we assume a fairly general continuous stream processing model in which a continuous query is expressed as a graph of operators interconnected via streams between input and output queues. As shown in Figure 1 operators receive tuples in their input queues (1), process those tuples (2), and preserve results in their output queues (3) until receiving an acknowledgment from all downstream nodes. Each stream is associated with an information schema describing the tuples that flow through it. Each operator implements a function such as filter, union, aggregate, join, etc. and may or may not be accumulating state over time. The latter distinction separates operators into stateless (such as filter, union) and stateful (such as aggregate, join) ones. Operators are executing in stream-processing engines (SPEs) that are distributed and communicating over the network.

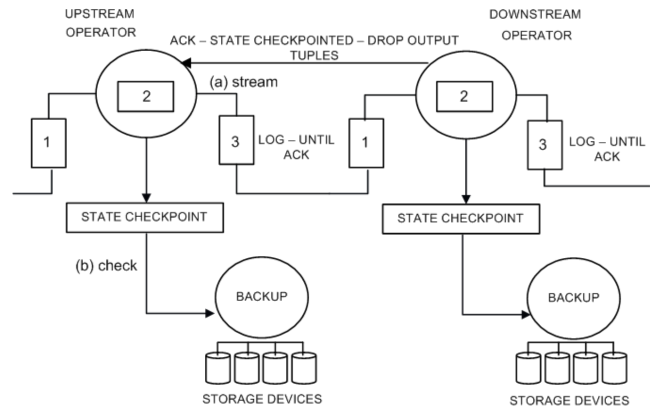

Figure 1. Checkpoint roll-backward (CRB).

In explaining the CEC technique we will use as a canonical example of a stateful operator the aggregate operator. The aggregate operator computes a function (such as count, minimum, maximum, sum, or average) on a specific tuple field over windows of eligible tuples (defined through another field). A typical example would be a query stating "Compute the average talk time of all telephone calls (described by CDRs) originating from a specific phone number over a window of 1 hour, grouped by phone number". The duration that a window remains open (and thus accumulates state) can either be time-based, as in the previous example, or count-based (remains open for a fixed number of tuples). The specification of the aggregate operator allows overlap between successive incarnations of a window (*i.e.*, advancing the window for less than its size) taking tuples from the previous incarnation into account when computing the next result. In practical implementations (as for example in Borealis [3][8]) such windows are implemented as two separate physical windows, both accumulating tuples during the overlap period, and thus do not present additional challenges compared to non-overlapping windows.

In what follows we briefly describe the general principles of the checkpoint roll-backward technique in Section II.A and then move into the details of the CEC methodology in Section II.B. A brief description of the stream persistence architecture that we developed for logging output tuples and CEC checkpoints to storage appears in Section IX (see [20] for a full paper).

A. Checkpoint roll-backward

Figure 1 provides an abstract view of the checkpointing procedure in CRB. Downstream operators are taking checkpoints of their state (b) and send acknowledgments to upstream queues that they no longer depend on the corresponding input tuples to reconstruct it. In the standard implementation of CRB the operator state is a point-in-time consistent view of all open windows at checkpoint-capture time. The full checkpoint includes the state of the operator itself and its input and output queues. We call this an instant checkpoint of the operator. For the rest of the paper we assume without loss of generality that only output queues need to be persisted while input queues can be rebuilt by fetching tuples from the upstream operators' output queues. Checkpoints are preserved in stable store (another node's main memory or a storage device).

The recovery procedure for a streaming node has two parts. First, one has to decide what the most recent checkpoint is; then ask upstream nodes to replay tuples from that point forward. If a node is asked to replay tuples that it has to reconstruct (perhaps because it also has failed and lost recent state), then upstream nodes have to go through the recovery process themselves. Note that the operator's internal state (open windows before failure) has to be checkpointed in-sync with its output queue.

B. Continuous eventual checkpoints

CEC departs from the standard implementation of CRB by breaking the overall operator state into independent components corresponding to the different open windows w_i and performing checkpoints of each of them asynchronously at different times t_i. Figure 2 depicts an example of an aggregate operator with N open windows at time t_c. The time is defined by the timestamp of the last tuple that entered the operator and affected its state by either updating an existing window, or opening a new window, or closing a window and emitting an output tuple.

Window checkpoints are expressed as control tuples containing the partial state S accumulated by the operator for a given window up to time t. Each window checkpoint additionally contains the following parameters:

1. Type of checkpoint (G^{check} or G^{open}):

- G^{check} is the type of checkpoint produced for an existing window. For a G^{check} checkpoint, S is equal to the partial result accumulated by the window up to time t.

- G^{open} is the type of checkpoint produced when a window opens. For a G^{open} checkpoint, S is equal to the state of the window after taking into account the tuple that opened it. Tuples of type G^{open} are necessary to ensure we have a guaranteed known state for a window at time t_c to roll back to in case a G^{check} has not been produced for it yet.

2. Total number of open windows (N) at checkpoint time t.

Figure 2 depicts the time evolution of an operator from time t_k to t_c with checkpoints for windows w_k, w_i, w_j, w_c. Standard data-carrying tuples produced when a window closes are abbreviated as R (result). G^{check}_k and R_k refer to a checkpoint and a data-carrying tuple for window w_k.

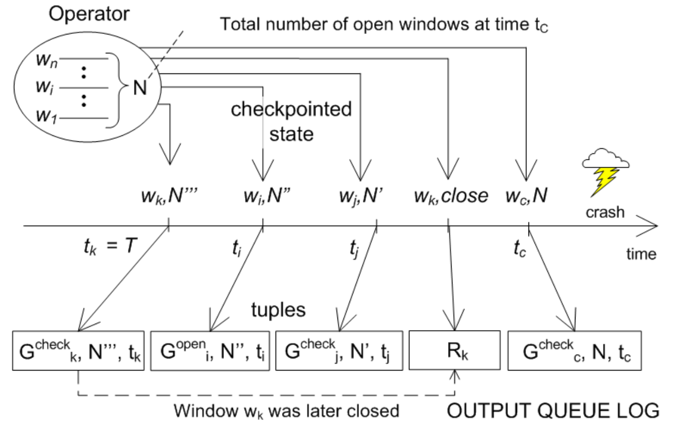

Figure 2. Continuous eventual checkpointing (CEC).

Window checkpoint tuples are logged at the operator's output queue along with regular output tuples in timestamp order. Thus G^{check}_k is emitted and stored in the log before the subsequent R_k produced when w_k closes (Figure 2). If a persistence mechanism is used to write the output queue to stable storage it should preserve timestamp order for all tuples produced by the operator.

In CEC the operator checkpoint is not a single, cohesive entity as in traditional CRB schemes. Instead, CEC maintains an **eventual checkpoint** (EC) at time t_c as a set of window checkpoints $W = \{(w_i, t_i): t_i \leq t_c\}$ that can be used to bring the operator to a consistent state, *i.e.*, a state the operator went through in the actual execution, at time t_c. This state includes all windows that were open at time t_c and for which, their most recent persisted "footprint" (*i.e.*, partial checkpoint) was written to stable storage prior to t_c. Once we determine the oldest "footprint" of any window that is open at t_c (and call its timestamp T, $T = \min_i (t_i)$ for all $t_i \leq t_c$) CEC loads onto the operator the state of all open windows w_i. This state corresponds to different times t_i and is thus not immediately consistent. To achieve consistency, the operator must contact its upstream node and request replay of input tuples with timestamps $t > T$. Since T is the earliest among all t_i's whereas the state of all (except one) windows reflects a later time, the operator will unavoidably see tuples that it has already seen in the past. To ensure that we reach the correct pre-failure state, the operator must ensure that window w_i ignores tuples with timestamps $t < t_i$.

Constructing an EC

A key challenge when constructing an eventual checkpoint from the persisted output queue is finding out how far to roll back into the log looking for window checkpoints (G^{open}, G^{check}). We solve this problem by storing along with each checkpoint the number *N* of open windows at that time, which turns out to be the number of different window "footprints" we need to look for when rolling back the log. For example Figure 2 shows that the checkpoint of w_c at time t_c records the total number of open windows *N* at that time. It is important to ensure that the entire checkpoint record (*S*, $G^{\{check|open\}}$, *N*, and *t*) is atomically written to stable storage. We ensure this by using a fingerprint to detect partial I/O errors and in that case discard the record.

During failure-free operation, CEC increases *N* when a window opens and decreases it when a window closes. By CEC rules, at time t_c all open windows must have produced a G^{open} and possibly G^{check} checkpoints in the output queue. During reconstruction it is important to skip G^{open} or G^{check} tuples for windows that are known to have closed prior to t_c. If a window closed prior to t_c (as for example is the case for window w_k in Figure 2), its R tuple will be encountered prior to any checkpoints for that window and therefore it will be excluded from the EC.

Extent of an EC

Our system evolves the eventual checkpoint continuously over time as new control and regular output tuples are being produced. Recall that for an eventual checkpoint W = $\{(w_i, t_i): t_i \leq t_c\}$, T is the earliest timestamp of any open window among the w_i in W. We define as the *extent* of W (henceforth referred to simply as the extent) to be the set of tuples (of any type, G^{open}, G^{check}, or R) in the output queue that recovery needs to go through when rolling-back to reach the tuple with timestamp T.

The extent is an important parameter in constructing the EC since the time to complete the construction is proportional to its size. Intuitively the rate of growth of the extent is inversely proportional to the rate of progression of T. T may be lagging behind in the past when the window it corresponds may have had no G^{check} tuple produced for it since the last G^{check} or G^{open} tuple produced for the same window. Besides increasing EC reconstruction time, a large T means that we will need to replay a large amount of tuples from upstream queues to bring the operator to a fully consistent state.

Controlling the extent

Windows can remain open for a long time mainly due to two reasons: (1) the stream tuple distribution favors some windows over others; (2) the window specification allows for a large accumulation of tuples before closing and computing a result, either via the tuple count or time parameters. We refer to windows that are staying open for long periods of time (due to (1), (2) or both) as *slow* and those that close and re-open frequently as *fast*. Note that depending on the characteristics of the incoming stream a fast window may turn into a slow one over time and vice versa.

The existence of slow windows in the operator state is a key factor leading to a growing extent size. Our goal in CEC is to produce G^{check} checkpoints more aggressively –as far as performance and recovery-time objectives allow— for the slowest windows in order to advance T and reduce the size of the extent.

CEC benefits

A benefit of performing individual window checkpoints is that we avoid freezing the operator for long checkpoint-capture time intervals typical of traditional CRB approaches. CEC still needs to devote time to individual window checkpoints, but this time is smaller, spread over a longer time period, and adjustable to application needs. Additionally we do not require any operating system support for copy-on-write or other memory-protection schemes that are typically used in traditional implementations of CRB, nor incur the overhead of protection violation exceptions in these schemes.

CEC enables a performance vs. recovery time tradeoff by parameterizing how frequently it produces checkpoints as well as the set of windows the checkpointing effort focuses on. As described, CEC focuses on checkpointing of slow windows. However the degree of intensity at which CEC is producing checkpoints may impact performance. The CEC performance vs. recovery time tradeoff is thus enabled through explicit control of the following parameters: (1) how much time to devote (out of the overall execution time) to checkpointing; (2) when is checkpointing necessary for the slowest windows. Section III provides details into the specific choices we have made in our implementation when tuning those parameters.

Finally, another benefit of CEC is that by integrating checkpoints into standard operator output it can leverage a single persistence architecture and stable storage structure for both operator and output-queue states. Note that although control tuples mix with data-carrying tuples in the stable storage abstraction, they do not complicate processing in downstream SPEs as the SPEs are able to recognize them in their input streams and disregard them during operator processing.

CEC challenges

Constructing an EC requires reading the output queue log sequentially looking for all window checkpoints that comprise the EC. This is a sequential process that can be sped up by reading large chunks (currently 256KB) of the log into memory to avoid the penalty of small I/Os. A factor that affects performance has to do with the way tuples are grouped in the output stream. The degree of grouping tuples into a structure called a *stream event* [8] before storing them in the distributed file system is proportional to the operator's output rate: operators with low output rate are expected to feature a smaller degree of grouping (in some cases, each tuple occupies a separate stream-event). In such cases a larger number of stream events to process in the recovery path will lead to a longer time to reconstruct the EC.

G^{open} checkpoints required by CEC on every window opening are expected to increase the amount of tuples produced by the operator per window from one to two (G^{open} at open and R on close), a fact that becomes more important when the application features a large number of fast windows. However this is not expected to be an issue in practice in most real deployments as stateful operators typically have low output tuple rates and consequently are much less I/O-intensive compared to stateless operators (a filter or a map) that produce an output tuple for each input tuple that they receive.

III. IMPLEMENTATION

In this section we describe the implementation of CEC using the aggregate operator as a case study. We use the Borealis [8] implementation of the operator as a reference but our principles are more general and can apply equally well to other continuous-query data stream processing systems.

First we describe the state maintained by the streaming operator. This state includes (a) the set of all currently open windows; (b) the state of each window: open or closed; accumulated state so far; and two timestamps τ_1 and τ_2, τ_1 corresponding to the input tuple that created the window and τ_2 corresponding to the last checkpoint (G^{open} or G^{check}) produced for the window; and (c) an ordering of all open windows by their τ_2 timestamp. The objective of this ordering is to always be able to start from the window with the least-recent checkpoint when checkpointing with the objective to reduce the size of the extent.

CEC requires minimal changes to the standard Borealis tuple header to store its own information. It uses the existing *type* attribute to indicate tuple type (G^{open}, G^{check} or R). The existing *timestamp* attribute is used to store the timestamp of the tuple that was last processed by the operator (t_{now}). Finally, CEC introduces a *count* field to indicate the number of active windows at tuple production time. These modifications work for both stateful and stateless operators since the later can be thought of as a special case of the former with window count 0.

In terms of execution paths we distinguish between foreground, background, and recovery. All processing is performed in the main thread of the operator because a separate thread would result in unnecessary complexity and locking overhead. The foreground path focuses on processing the tuples entering the operator through the input stream. The last tuple that entered the operator may be either (1) opening a new window; (2) contributing to an existing window; (3) contributing to an existing window and causing its closure. In case (1) we increase the counter of active windows N, set τ_1 and τ_2 to t_{now} and produce a G^{open} tuple into the output stream. The G^{open} tuple carries the number N, the current state of the window, and the timestamp t_{now}. In case (2), the tuple updates the state of a window; τ_2 is not updated until the first G^{check} emitted for that window. The number of active windows remains unchanged. In case (3) the counter of active windows is decreased, an R tuple is produced, and the just-closed window is erased from the ordered list of open windows of point (c) above.

In the background path the operator periodically checkpoints slow windows via the production of G^{check} tuples. The G^{check} tuples are marked with the t_{now} timestamp of the tuple entering the operator at the time of G^{check} production. Simultaneously the τ_2 timestamp for the corresponding window changes to t_{now}. The number of windows eligible for checkpointing at any time (we call this the *intensity* of checkpointing) is decided based on recovery-time objectives expressed by two parameters Q, U described in Section III.A. To avoid the impact of uncontrolled checkpointing on response time we limit checkpointing only to specific time intervals. We use two parameters to denote the amount of time we devote to checkpointing (*checkpoint interval,* CI) and the time between checkpointing intervals (*checkpoint period*, CP). The combination of Q, U and CI/CP can be used to adjust to a desirable performance vs. recovery speed operating point as we demonstrate in our evaluation section. Note that the choice of these parameters does not affect correctness because even if we delay checkpointing, a window will roll-back to an older G^{check} for that window or –in the worst case— to the G^{open} tuple for that window. at the expense of longer recovery time.

During the recovery path, CEC reconstructs the eventual checkpoint by sequentially rolling back on the output queue log as described in Section III.B. Following EC reconstruction, the EC must be loaded onto the operator to form its new state prior to asking upstream nodes for tuple replay. To load the EC onto the operator in an as simple manner as possible we created a special input stream (in addition to the standard input stream of the operator) through which we feed the EC into the operator. We call this new stream the "EC-load" stream. A complication we had to address with "EC-load" is that although it is created as an output stream (following the output tuple schema) it must be fed to the operator through an input stream. We further modified the operator threads to be aware that tuples coming from "EC-load" can only be part of the EC for the purpose of recovery. The implementation of the recovery code uses support provided by our persistence architecture described briefly in the Appendix and in [20]. The communication protocol between a recovering node and its upstream and downstream nodes (synchronization with upstream to request replay from given timestamp, synchronization with downstream to find out what it wants to have replayed) builds on the RPC message exchange framework provided by Borealis.

A. Policies for producing G^{check} tuples

Recall that CEC maintains an ordered list of open windows by their τ_2 time of last checkpoint (G^{new} or G^{check}), keeping the window with the oldest checkpoint at the top. This checkpoint is by definition at the end of the extent and thus the corresponding window (w_k in the example of Figure 3) is the prime candidate to produce a new checkpoint for (thus decreasing the size of the extent). To maintain an accurate estimate of the current size of the EC extent as well the amount of tuples that need to replayed by upstream

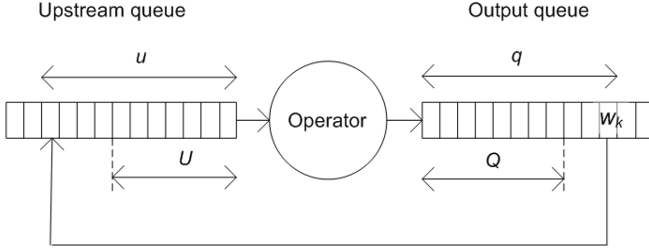

Figure 3. Extent size (q), number of upstream tuples to replay for given extent (u), and corresponding targets (Q, U).

operators during recovery, we maintain the following state: For each window w_i in the open-window list we store (a) the number of tuples emitted by the operator between w_i and the previously checkpointed window; and (b) the number of input tuples processed by the operator between producing checkpoints for these windows. Effectively, (a) provides a measure of the size of the extent, which we call q, and (b) provides a measure of the number of tuples u we need to replay where a crash to occur at this point in time. We have devised two methods to determine when it makes sense to produce a checkpoint for the oldest window: a method based on a cost-benefit analysis and another based on explicitly setting targets for q, u.

The first method considers the costs and benefits of checkpointing: First, taking a G^{check} has the benefits of reducing the size of the extent (and thus the cost of eventually constructing the EC); second, a G^{check} brings forward the timestamp of the first tuple to replay, thus reducing the cost of replay. On the other hand, taking a G^{check} has two costs: First, it adds another tuple to the extent, increasing its size by one; second, it incurs the overhead (CPU and I/O) of producing the G^{check}. Based on the above it is reasonable to only perform a G^{check} when the benefits outweigh the costs. It is straightforward to derive an analytical cost-benefit formula based on the above principles but one needs to calibrate it for a given platform by including a number of empirically-measured parameters. The full implementation and evaluation of this method is an area of ongoing work.

The second method (which is used in our experiments) takes the approach of explicitly setting appropriate empirically-derived targets for q and u. For example the policy "produce a G^{check} if $q > Q$ or $u > U$ or both, where Q = 1,000 and U = 1,000,000" means that if the extent or the number of tuples to replay exceeds Q, U respectively, then try to decrease them by taking checkpoints starting from the older end of the extent (w_k in Figure 3). Notice that values of Q should typically be smaller than values of U reflecting the fact that a tuple carries a heavier weight when considered in the context of EC reconstruction than in stream replay (to say it simply, a tuple costs much more to process in EC reconstruction than in stream replay).

B. Implementation complexity

The overall modifications to Borealis to support CEC are about 700 lines of code in our persistence and recovery mechanisms and about 30 lines of new code in the implementation of the aggregate operator. We have also added minor modifications to the Borealis SPE and consumer application code to drop G^{open} and G^{check} tuples upon reception. We disable this feature in our experiments in order to be able to measure the number of checkpoint tuples produced as well as the aggregate throughput (control plus data) observed by the final receiver.

IV. Evaluation

Our experimental evaluation of CEC focuses on three key areas: (1) The impact of CEC on streaming performance when operating under minimum recovery-time guarantees (henceforth referred to as baseline performance); (2) the impact of a range of Q, U values to operator recovery time; and (3) the response time vs. recovery time tradeoff with varying CI/CP. Our experimental setup consists of three servers as shown in Figure 4. All servers are quad-core Intel Xeon X3220 machines with 8GB of RAM connected via a 1Gb/s Ethernet switch.

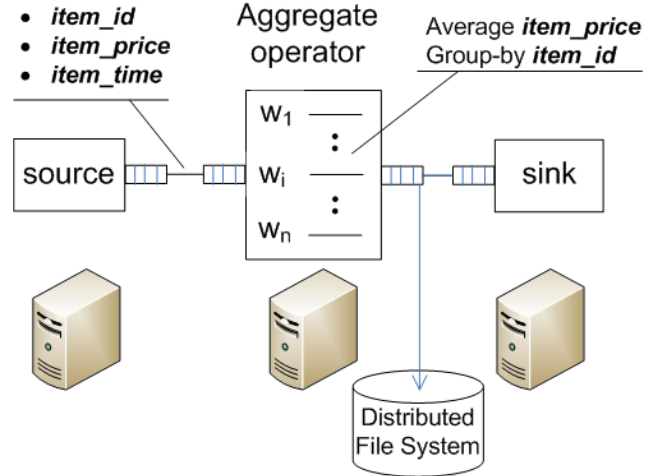

Figure 4. Experimental setup.

The first server hosts the tuple-producing engine (or source). The tuples produced consist of three fields: *item_id*, *item_price*, and *item_time*, where *item_id* is an integer identifying the item (e.g., an SKU), *item_price* is an integer indicating the item's price, and *item_time* represents the time of purchase of the item. The tuple size is 100 bytes. The second server hosts an aggregate operator computing the average purchase price of items grouped by *item_id* (in other words the operator will maintain a separate window per *item_id* computing the average over a number of tuples equal

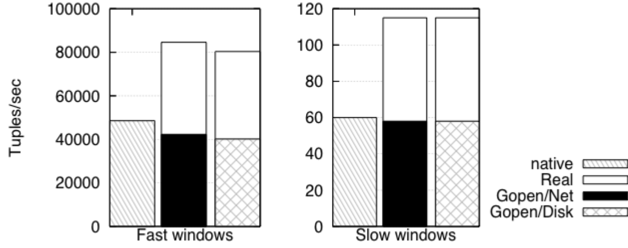

Figure 5. CEC baseline impact (G^{open} tuples only).

to the window size). The third server hosts the tuple consumer (or sink).

The window size of the aggregate operator in all experiments is defined based on number of tuples received (count-based windows). We chose to evaluate CEC with count-based windows rather than time-based ones due to the special challenges posed by the former. The amount of time count-based windows remain open depends strongly on the distribution of input tuples and can be indefinite. Our results hold for time-based windows as well as for other stateful operators. Finally, we use a distributed file system (PVFS [18]) to persist the output queue of the aggregate operator.

A. *CEC baseline impact*

We first highlight the impact of CEC while operating under minimum recovery-time guarantees. In this case CEC produces G^{open} tuples (necessary for correctness) but no G^{check} tuples (necessary to reduce the extent). We configure the aggregate operator for two different workloads: *fast-windows* and *slow-windows*. The fast-windows workload uses a window size of 1. Each input tuple entering the operator forces the creation of a new window and its instant closure, emitting an R tuple. The slow-windows workload uses a window size of 1000. In both cases the source produces tuples randomly distributed across two *item_id*s. In all experiments the rate at which the tuple-generator injects tuples to the network is limited by its CPU or by network flow control.

Figure 5 illustrates the performance impact of CEC on streaming throughput under the two workloads described above. The results are compared against those of native Borealis in the same setups. Other configurations depicted in Figure 5 include: *Gopen/Net*, which is the CEC setup where the aggregate operator forwards G^{open} tuples to the network but does not persist them at its output log; and *Gopen/Disk*, which includes the additional overhead of persisting G^{open} tuples. Gopen/Net and Gopen/Disk are measured at the sink after dropping control tuples. The Real bar represents the real throughput that our receiver observes including both normal (R) and control (G^{open}) tuples.

A key observation is that order-of-magnitude differences in output rate across workloads are mainly due to different operator specifications (higher window counts result in lower output rates). Performance of the Gopen/Net configuration drops by less than 10% versus native performance across all workloads. This decrease can be attributed to the ratio of G^{open} vs. R tuples injected to the output stream: wider-spaced R tuples translate into lower R-tuple rate at the sink. Performance of the Gopen/Disk configuration is nearly identical to Gopen/Net, indicating that the I/O path has minimal impact on throughput, especially on lower output rates. The Real bar shows that the actual throughput seen by the receiver is twice the throughput of data tuples alone.

In terms of CPU usage, native Borealis consumes about 150% (fast-windows) and 130% (slow-windows) out of a total capacity of 400%. With CEC (G^{open} tuples only) the CPU utilization remains the same for the Gopen/Net configuration and increases in the Gopen/Disk setup to 200% for the fast-windows workload. This increase can be attributed to the overhead of the I/O path. Throughout these experiments we did not saturate the CPU or the network at the server hosting the aggregate operator. The main factor limiting performance is the degree of parallelism available at the tuple source (Borealis does not allow us to drive a single aggregate operator by more than one source instances).

B. *CEC recovery time*

Next we focus on CEC recovery time using a range of Q and U values. Recall that Q represents the extent size to exceed before checkpointing the oldest window, whereas U represents the number of upstream tuples (to replay) to exceed before checkpointing the oldest window. In all cases the source produces tuples randomly distributed across 100,000 *item_id*s. The aggregate operator uses a window size of 10 tuples. To isolate the impact of each of Q, U on recovery time we set up two separate experiments with a different parameter regulating recovery time in each case. The extent size is measured at a time of crash in the middle of each run. Our operator graph in this experiment differs from that depicted in Figure 4 in that we interpose a filter operator between the source and the aggregate to be able to persist and replay the filter's logged output tuples during recovery. The CI parameter is set to 5ms and the CP to 100ms in all cases.

Figure 6 depicts the effects of varying Q from 1 to 8 times the number of open windows (openwins) on two recovery-time metrics: eventual checkpoint (EC)-load time; bytes replayed by upstream source; and two CEC-internal metrics: extent size; and number of G^{check}s emitted. The average value of openwins in the operator is measured to be around 90,000. Our smallest Q value (1*openwins) represents the extent size that the operator would have produced by just emitting G^{new} tuples. This is the smallest possible extent for that number of windows. Setting Q to anything less than that would be setting an unattainable target, resulting in unnecessary overhead.

Figure 6 (upper left) depicts the number of G^{check} tuples produced as the extent is allowed to grow larger. An observation from this graph is that the number of G^{check}s drops sharply for Q > 2*openwins, evidence that the extent in these cases nearly always stays below target. Figure 6 (upper right) depicts the extent size with growing Q. CEC cannot achieve the target of Q = 1*openwins due to the stringency of that goal but manages to achieve it in the Q=2, 4*openwins cases. The extent size for Q = 8*openwins (500,000) is lower than its targeted goal (720,000) due to the

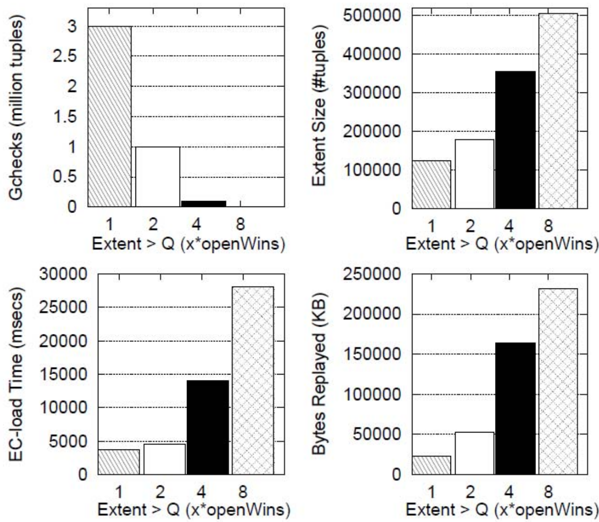

Figure 6. Effect of Q.

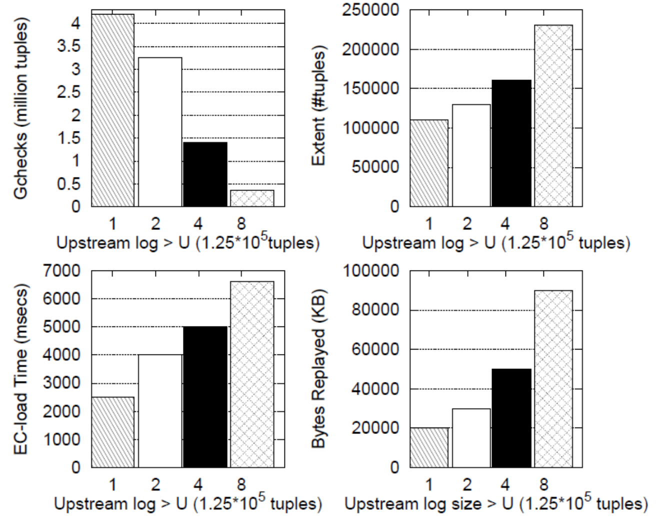

Figure 7. Effect of U.

fact that 500,000 is the maximum extent size reachable in this operator configuration.

Figure 6 (lower left) depicts the increase in EC-load time with growing Q due to the operator having to read and process a progressively larger extent to reach its pre-failure state. We observe that for a smaller extent (120,000 and 180,000 tuples) the time to read and load it into the operator varies from 3.5 to 5 seconds. A larger extent leads to EC-load times of between 14 and 27 seconds. Figure 6 (lower right) depicts the number of bytes to replay from the upstream node after the operator has loaded the EC. Our results here exhibit the same trends as in EC-load. Lower Q values correspond to replaying 25MB to 50MB of tuples, whereas higher Q values correspond to replaying more than 200MB of tuples. The CPU performing the EC-load operation was always 100% busy. CPU overhead due to CEC during failure-free operation is minimal.

In the U experiment we vary the values of U between 125,000 and 10^6 tuples. Figure 7 illustrates the impact of U on the same four metrics. Figure 7 (upper left) shows that relaxing the U target leads to fewer G^{check} tuples produced, consistent with our expectations. The drop is not as sharp as in the Q experiment, evidence that our large Q values lead to more relaxed policies compared to large U values in this experiment. Figure 7 (bottom right) shows that higher values of U result to between 20MB (200,000 tuples) and 90MB (900,000 tuples) of upstream input replayed during recovery.

Figure 7 (upper right) shows that the extent size increases from 120,000 tuples to 230,000 tuples. This increase can be explained by the lower production of G^{check} tuples with growing U. Consistent with having to handle longer extents, EC-load time (bottom left) increases from 2.5s to 6.7s. The EC-load times measured for different extent sizes are in agreement with our results in the Q experiment.

C. *Impact of checkpointing period*

Next we evaluate the impact of different values of CI/CP to operator response and recovery time. In this experiment we vary CP between 25ms and 250ms while maintaining CI fixed at 10ms. All runs use the aggregate operator configured as described in the previous section. Q is fixed and equals the number of open windows (whose average was measured to be about 90,000), a very aggressive target that ensures there always exist windows that are candidates for checkpointing.

Figure 8 depicts the response time of the aggregate operator measured as the difference between the opening and closure time of a window, reported as per-second averages. Operator response time ranges from 16.5s (without CEC), to 18.5ms, 21.5ms, and 32.5s (CEC with CP = 100ms, 50ms, 25ms, respectively) reflecting the fraction of operator time spent on checkpointing. For CP = 250ms (not shown in Figure 8) CEC response time approximates that of native (no CEC). To highlight the tradeoff between response time and recovery time, in Figure 9 we report the extent size as a function of CP. Longer CP values result in longer extent sizes and consequently longer recovery times. Based on our results from the Q, U experiments we estimate that varying CP from 25ms to 250ms increases EC-load time from 2s to 5s. Given the higher impact of other factors such as failure detection, RPC communication, etc. in overall recovery time it is preferable in this particular setup to choose the most relaxed checkpoint period (e.g., CP = 250ms) achieving response time close to that of performing no checkpointing at all.

V. DISCUSSION

Although we have demonstrated CEC primarily for the case of an aggregate operator, we believe that CEC can be also applied to other stateful operators such as *join*. Join [3] has two input streams and for every pair of input tuples applies a predicate over the tuple attributes. When the

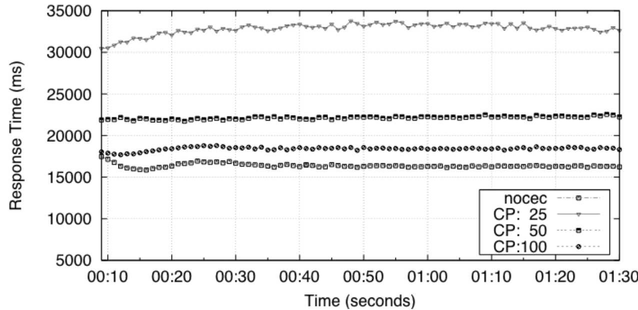

Figure 8. Operator response time for different checkpoint periods.

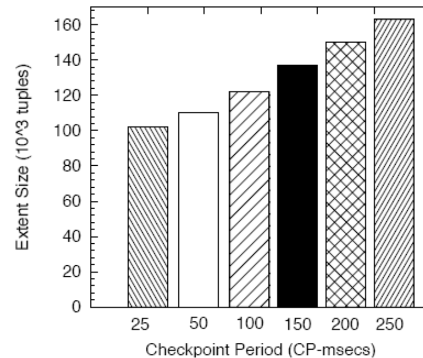

Figure 9. Extent size for different checkpoint periods.

predicate is satisfied, join concatenates the two tuples and forwards the resulting tuple on its output stream. The stream-based join operator matches only tuples than fall within the same window. For example assume two input streams, R and S, both with a time attribute, and a window size, w. Join matches tuples that satisfy t.time -s.time ≤ w, although other window specifications are possible. Tuples produced by operators in most cases maintain timestamp ordering or else ordering can be applied using sort operators during insertion. If output queues are persistent, then the timestamp of the result tuple can be changed to resemble the timestamp of the concatenated tuples. This way, in case of failure, the operator knows from which point in time to ask replay from both input streams. In addition, in join operator we have to remember the relevant position inside the input streams from which concatenation takes place. CEC will periodically produce G^{check} tuples indicating the relevant position of the two windows at any point in time.

VI. Related work

General fault-tolerance methods typically rely on replication to protect against failures. The main replication mechanisms for high-availability in distributed systems include *state machine* [19], *process-pairs* [11], and *rollback recovery* [10] methodologies. In the state-machine approach, the state of a processing node is replicated on *k* independent nodes. All replicas process data in parallel and the coordination between them is achieved by sending the same input to all replicas in the same order. The process-pairs model is a related approach in which replicas are coordinated using a primary/secondary relationship. In this approach a primary node acts as leader forwarding all of its input to a secondary, maintaining order and operating in lock-step with the primary node. In rollback recovery, nodes periodically send snapshots (typically called checkpoints) of their state to other nodes or to stable storage devices. Upon recovery, the state is reconstructed from the most recent checkpoint and upstream nodes replay logged input tuples to reach the pre-failure state. All of the above methodologies have in the past been adapted to operate in the context of continuous-query distributed stream processing systems.

Two examples of the state machine approach adapted for stream processing are active-replicas [6] and Flux [21]. Both systems replicate the producer and the consumer operators in a stream dataflow graph in a symmetric fashion. Each consumer replica receives tuples from one of the producer replicas and, in case of producer failure, the consumer switches to another functioning producer replica. Strict coordination is not required since consistency is eventually maintained by the replicas simultaneously processing the same input and forwarding the same output. All operators preserve their output queues, truncating them based on acknowledgements periodically sent by consumers. In case of failure, all upstream replica nodes are up-to-date and can start serving their downstream nodes as soon as the failure is detected, minimizing recovery time.

The work of Hwang et al. [14] extends the active-replicas approach so that all upstream replicas send their output to all downstream replicas and the latter being allowed to use whichever data arrives first. Since the downstream nodes receive data from many upstreams, the input stream received might be unordered or/and contain duplicate tuples. Despite the above complications their system manages to deliver the same result as it would produce without failures. To achieve this, operators are enhanced with extra non-blocking filters (one filter per input stream) that eliminate duplicates based on periodically exchanged timestamp messages *t*. All tuples with timestamp lower than *t* are considered duplicates and dropped.

Another fault-tolerance methodology that combines the active-replicas and process-pair approaches is *active standby* [13]. In active-standby, secondary nodes work in parallel with the primary nodes and receive tuples directly from upstream operators. In contrast to active-replicas, in active-standby secondary nodes log result tuples in their output queues but do not forward tuples to secondary downstream neighbors. Challenges with this approach include output preservation due to non-deterministic nature of operators and bounding the log of each secondary.

Instances of the rollback recovery (also known as checkpoint-rollback or CRB) methodology [10] are the so-called *passive-replicas* approaches, comprising *passive-standby* and *upstream-backup* [6][13]. In passive-standby, the primary replica periodically produces checkpoints of its state and copies it to the backup replica. The state includes the data located inside the operators, along with the input and output queues. The secondary node acknowledges the state

already received with the primary upstream so as to drop tuples from the latter's output queue. In case of failure, the backup node takes over by loading the most recent checkpoint to its current state. A variant of passive-standby that allows independent checkpointing of fragments (sub-graphs) of the entire query graph has been shown [15] to reduce the latency introduced by checkpointing. However checkpoint granularity with this methodology is still at the level of entire operators and stream processing freezes while storing a fragment checkpoint to a remote server memory.

The upstream-backup [13] model was proposed for operators whose internal state depends on a small amount of input. In this approach, the upstream nodes act as backups for the downstream nodes by logging tuples in their output queues until all downstream nodes completely process their tuples. The upstream log is trimmed periodically using acknowledgments sent by the downstream primaries. In case of failure, the upstream primaries replay their logs and the secondary nodes rebuild the missing state before starting to serve other downstream nodes. In contrast to passive-standby, upstream-backup requires a longer recovery but comes with lower runtime overhead.

In all aforementioned methodologies replica nodes retain output tuples and checkpoints in memory buffers reducing the amount memory available for input tuple processing. One solution to this problem is to utilize persistent storage [15][20]. SGuard [17] is a system that leverages the use of a distributed and replicated file system (HDFS [7]) to achieve stream fault-tolerance in Borealis [8]. Operators periodically produce delta-checkpoints of their current state and the recovery is made using the latest checkpoint of the failed node. In this approach, HDFS act as the backup location for the checkpointed state, thus reducing the memory requirements of the stream processing nodes. To eliminate the overhead of freezing the operators during checkpoint, SGuard performs checkpoints asynchronously and manages resource contention of the distributed file system with the enhancement of a scheduler that batches together several write requests. SGuard is related to CEC in its focus on producing operator checkpoints and persisting them on stable storage. SGuard however considers the entire operator as a checkpoint unit whereas our approach breaks operator state into parts, treating each window as an independently entity. Another system that takes advantage of operator semantics to optimize checkpointing performance is SPADE [16].

Zhou et al. [22] use log-based recovery and *fuzzy* checkpointing to offer programming support for high-throughput data services. Their fuzzy checkpoints of independent memory objects are similar to our eventual checkpoints of independent operator-window states and their logs are similar to our upstream queues, which can replay input tuples during recovery. In addition, they propose an adaptive control approach to regulating checkpoint frequency based on a number of target parameters. Our work differs in that our checkpoints are integrated within the logging infrastructure; they are continuously and incrementally evolving; and we use a different set of target parameters to regulate checkpointing intensity.

VII. Conclusions

In this paper we proposed a new methodology for checkpoint-rollback recovery for stateful stream processing operators that we call continuous eventual checkpointing (CEC). This novel mechanism performs checkpoints of parts of the operator state asynchronously and independently in the form of control tuples produced by the operator. Individual window checkpoints are interleaved with regular output tuples at the output queue of each operator and persisted in stable storage. During recovery, CEC processes the output queue of the operator to reconstruct a full checkpoint, which it then loads on the operator. The checkpoint determines the amount of tuples that need to be replayed by the upstream source. Our results indicate that CEC does not penalize operator processing when operating under minimal recovery guarantees. Offering stronger recovery guarantees is possible through tuning of the Q, U target parameters regulating the eventual checkpoint's extent size and upstream queue replay size. The checkpoint interval and period parameters CI, CP can further tune the system to the desired response-time objective. Overall our results demonstrate that CEC is a simple to implement, configurable, low-overhead checkpoint-rollback solution for mission-critical stream processing operators.

VIII. Acknowledgments

We thankfully acknowledge the support of the European FP7-ICT program through the STREAM (STREP 216181) and SCALEWORKS (MC IEF 237677) projects.

IX. Appendix: Persistence Architecture

CEC's intermixing of operator state checkpoints with regular output tuples at the operator's output queue raises the need for a queue persistence mechanism in the context of a stream processing engine (SPE). The upper-left part of Figure 10 shows a standard SPE structure such as found in the Aurora/Borealis [4] system. In this structure incoming tuples from a stream S are shepherded by a thread (the Enqueue thread in this figure) to be enqueued into the SPE for processing by operator(s) that use S as one of their inputs. The tuples produced by the operators are placed on an output stream and dequeued by a separate thread (the Dequeue thread in the figure) and grouped into stream events (SEs). SEs are serialized objects grouping several tuples for the purpose of efficient communication. The remainder of the figure describes the path that persists SEs prior to communicating them to downstream nodes.

Our detailed description starts with the case of failure-free operation:

Failure-free operation: SEs created by the dequeue thread are first serialized. If the streams they are associated with are set for persistence (on a per-operator basis through a configuration option) the SEs enter the persist-event list, otherwise they move directly onto the forward-event list. A write operation to storage is initiated using a non-blocking API with asynchronous notification [2]. We additionally use checksums to check that an I/O has been performed correctly in its entirety on the storage device. The asynchronous I/O

operations are handled by a state machine in an event loop. For parallelism, we maintain a configurable window of N concurrently outstanding I/Os. Once a completion of a write I/O is posted by the storage system we first update a per-stream index (shown in Figure 10), and then move the persisted event data structure to the forward-event list. Subsequently a network send operation is initiated. The SE remains there until successfully sent out over the network.

The stream index maps a timestamp into a serialized SE that contains a tuple with that timestamp. The mapping is typically the file offset within the persisted object. In our current prototype the index is implemented using an Oracle Berkeley DB database.

Operation under failure: When a downstream SPE node fails, all streams connected to queues on that node disconnect and no outgoing network communication takes place on those streams until reconnection (other streams however are not affected). SEs produced by local operators are still persisted as described during failure-free operation. However, as soon as the SPE receives an I/O completion for an SE, it deletes it from memory. Other SEs belonging to still-connected streams proceed to the forward-event list as described during failure-free operation.

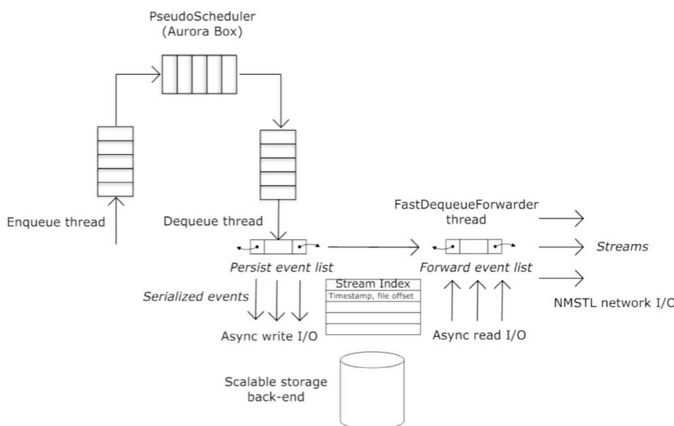

Figure 10. Stream processing engine I/O architecture.

Recovery: A recovering SPE node can reconnect to upstream SPEs serving the streams it was connected to prior to its failure. The recovering SPE performs the following steps: (1) reconcile the stream index in the DB with the log length reported by the file system; (2) determines the last consistent operator checkpoint, load the checkpoint, and determine the timestamp they want to start replaying from, as described in Section II.B; (3) communicate the timestamp to the appropriate upstream SPEs, which will replay tuples from those streams. Upon such a request from a downstream node SPE, an upstream node will look up the requested timestamp into its stream index. The lookup will return a pointer to an SE x that contains the requested timestamped tuple. The node will then start issuing asynchronous read I/O operations for stored SEs starting from x in a manner similar to the write operations described during failure-free operation. Upon completion of a read I/O, the retrieved SE may need to be de-serialized (if a subset of the SE is requested, as for example, in the beginning of a stream replay request) then re-serialized (if needed) and put into the forward-event list. Similar to the process followed during failure-free operation, the SE will be sent to the connected downstream SPEs over the network.

Catching up with a live stream: The persisted queue may be growing by simultaneously appending tuples to it (incoming on a live stream), and reading tuples by multiple clients from different offsets. In certain cases, the rate at which a reader consumes tuples may be higher than the rate at which tuples are produced (as for example when tuples are produced at a source-determined rate of a few Mbps while the reader consumes as fast as I/O resources possibly - several tens of MB/s- allow). In such cases the read pointer into the persisted queue may reach the end -in essence, exhausting the object portion that exists only in storage. Read I/O operations will then start being satisfied from memory buffers and we can say that the reader has "caught up" with the live stream. In such cases, an SPE may decide to interrupt stream persistence if the reason for it was to avoid tuple loss due to a downstream node failure. In cases where persistence has been explicitly requested, the two I/O directions (reads and writes) can continue to be simultaneously active with reads satisfied at memory speeds.

REFERENCES

[1] IBM Ushers in Era of Stream Computing. IBM Press Report, http://www-03.ibm.com/press/us/en/pressrelease/27508.wss.

[2] Kernel Asynchronous I/O for Linux, http://lse.sourceforge.net/io/aio.html.

[3] Borealis Application Programmer's Guide, Borealis Team, 2008.

[4] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. The VLDB Journal, 12(2):120–139, 2003.

[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. Springer, 2004.

[6] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. ACM Transactions on Database Systems, 33(1):1–44, 2008.

[7] D. Borthakur. The Hadoop Distributed Fle System: Architecture and Design, The Apache Software Foundation, 2007.

[8] Cetintemel, Abadi, Ahmad, Balakrishnan, Balazinska, Cherniack, Hwang, Lindner, Madden, Maskey, Rasin, Ryvkina, Stonebraker, Tatbul, Xing, and Zdonik. The aurora and borealis stream processing engines. In Data Stream Management: Processing High-Speed Data Streams, 2006.

[9] C. Drew. Military is Awash in Data from Drones, New York Times, January 10, 2010.

[10] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv., 34(3):375–408, 2002.

[11] J. Gray. Why Do Computers Stop and What Can be Done About it?, Tandem Technical Report 85-7, 1985.

[12] D. Hilley and U. Ramachandran. Persistent temporal streams. In Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, pages 1–20, New York, NY, USA, 2009.

[13] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In ICDE '05: Proceedings of the 21st International Conference on Data Engineering, pages 779–790, Washington, DC, USA, 2005.

[14] J.-H. Hwang, U. Cetintemel, and S. Zdonik. Fast and highly-available stream processing over wide area networks. In ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, pages 804–813, Washington, DC, USA, 2008.

[15] J.-H. Hwang, Y. Zing, U. Cetintemel, and S. Zdonik. A Cooperative Self-Configuring High-Availability Solution for Stream Processing. In ICDE '07: Proceedings of the 2007 IEEE 23th International Conference on Data Engineering, pages 176–185, Istanbul, Turkey, 2007.

[16] G. Jacques-Silva, B. Gedik, H. Andrade, K.-L. Wu, Language-level Checkpointing Support for Stream Processing Applications, in Proceedings of 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'2009), Lisbon, Portugal, 2009.

[17] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant Stream Processing Using a Distributed, Replicated Fle System. volume 1, pages 574–585. VLDB Endowment, 2008.

[18] M. Ligon and R. Ross. Overview of the Parallel Virtual Fle System. In Proceedings of Extreme Linux Workshop, 1999.

[19] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. ACM Computing Surveys, 22(4):299–319, 1990.

[20] Z. Sebepou and K. Magoutis. Scalable Storage Support for Data Stream Processing. In Proceedings of 26th IEEE Conference on Mass Storage Systems and Technologies (MSST 2010), Lake Tahoe, Nevada, May 2010.

[21] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly Available, Fault-Tolerant, Parallel Dataflows. In SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, New York, NY, 2004.

[22] J. Zhou, C. Zhang, H. Tang, J. Wu, T. Yang, "Programming Support and Adaptive Checkpointing for High-Throughput Data Services with Log-Based Recovery", in Proceedings of 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'2010), Chicago, IL, 2010.