Availability-Consistency Trade-offs in a Fault-Tolerant Stream Processing System

Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Mike Stonebraker
MIT Computer Science and Artificial Intelligence Laboratory
The Stata Center, 32 Vassar Street
Cambridge, MA 02139
Email: {mbalazin,hari,madden,stonebraker}@csail.mit.edu

ABSTRACT

This paper presents an approach to fault-tolerant stream processing. In contrast to previous techniques that handle node failures, our approach also tolerates network failures and network partitions. The approach is based on a principled trade-off between consistency and availability in the face of failure, that (1) ensures that all data on an input stream is processed within a specified time threshold, but (2) reduces the impact of failures by limiting if possible the number of results produced based on partially available input data, and (3) corrects these results when failures heal. Our approach is well-suited for applications such as environment monitoring, where high availability and "real-time" response is preferable to perfect answers.

Our approach uses replication and guarantees that all processing replicas achieve state consistency, both in the absence of failures and after a failure heals. We achieve consistency in the former case by defining a data-serializing operator that ensures that the order of tuples to a downstream operator is the same at all the replicas. To achieve consistency after a failure heals, we develop approaches based on checkpoint/redo and undo/redo techniques.

We have implemented these schemes in a prototype distributed stream processing system, and present experimental results that show that the system meets the desired availability-consistency trade-offs.

1. INTRODUCTION

In recent years, a new class of data-intensive applications requiring the "real-time" processing of large volumes of streaming data has emerged. These *stream processing applications* arise in several different domains, including computer networks (*e.g.*, intrusion detection), financial services (*e.g.*, market feed processing), medical information systems (*e.g.*, sensor-based patient monitoring), civil engineering (*e.g.*, highway monitoring, pipeline health monitoring), and military systems (*e.g.*, platoon tracking, target detection).

In all these domains, stream processing entails the composition of a relatively small set of time-oriented operators (*e.g.*, filters, aggregates, and correlations) on "windows" of data. In addition, most stream processing applications require results to be continually produced at low latency, in the face of high and variable input data rates. As has been widely noted [1, 8, 13], these features and requirements render traditional data base management systems (DBMSs) based on the "store-then-process" model inadequate for high-rate, low-latency stream processing.

Stream processing engines (SPEs) (also known as data stream managers [1, 30] or continuous query processors [13]) are a class of software systems that handle the data processing requirements mentioned above. Much work has been done on data models and operators [1, 5, 15, 28, 40], efficient processing [6, 7, 11, 30, 42], and resource management [12, 16, 30, 35, 37] for SPEs. Stream processing applications are inherently distributed, both because input streams often arrive from multiple geographically distributed data sources, and because running SPEs on multiple nodes enables better performance under high load [14, 35].

In this paper, we add to the body of work on SPEs by addressing *fault-tolerant stream processing*, presenting algorithms, implementation details, and experiments that enable distributed SPEs to cope with a variety of network and system failures. Our work differs from previous approaches [25, 35] to high-availability and fault tolerance in streaming systems in that those approaches can survive only the failure of processing nodes and are not tolerant to other types of failures, such as network failures, network partitions, and transient failures of input streams. Our approach addresses all of these types of failure in a single framework.

As in most previous work on masking software failures, we use replication [22], running multiple copies of the same query network on distinct processing nodes. Each SPE produces result streams that are either sent to applications or downstream SPE nodes for additional stream processing. When a downstream processing node stops receiving data (or "heartbeat" messages signifying liveness) from its upstream SPE node, it picks a different upstream replica (if it can find one) from which to obtain the missing input streams.

For a downstream SPE to be able to correctly continue its processing from a new upstream replica, the upstream replicas must all be consistent with each other. They must process their inputs in the same order, progress at roughly the same pace, and their internal computational state must be the same. To ensure replica consistency, we define a simple data-serializing operator, called *SUnion*, that takes multiple streams as input and produces one output stream with deterministically ordered tuples.

At the same time, if a downstream SPE is unable to find a suitable upstream data source for a previously available input stream, it must decide whether to continue processing with the remaining (partial) inputs, or block until the failure heals. If it chooses the former option, a number of "wrong" results will be produced, while the latter option greatly reduces availability because applications will see a partial failure as a complete one.

Our approach gives the user explicit control of trade-offs between consistency and availability in the face of network failures [10, 22] in an SPE. To provide high availability, each SPE guarantees that input data is processed and results forwarded within a user-specified time threshold of its arrival, even if some of its inputs are currently unavailable. At the same time, to prevent downstream nodes from unnecessarily having to react to unstable data, an SPE should try to avoid or limit the number of unstable tuples it produces.

We introduce an enhanced streaming data model in which results based on partial inputs are marked as *unstable*, with the understanding that they may subsequently be modified; all other results are considered *stable* and immutable. When a failure heals, each SPE that saw unstable data reconciles its state by re-running its computation on its correct and complete input streams. While correcting its internal state, the replica also *stabilizes* its output by *replacing* the previously unstable output with stable data tuples forwarded to downstream clients. We argue that traditional approaches to record reconciliation [27, 41] are ill-suited for streaming systems, and adapt two approaches similar to known checkpoint/redo and undo/redo schemes [17, 23, 22, 29, 38] to allow SPEs to reconcile their states.

Specifically, we address the problem of minimizing the number of unstable result tuples while guaranteeing that the results corresponding to any new tuple are sent downstream within a specified time threshold. This trade-off between availability (the specified threshold) and consistency (number of unstable result tuples, since that is often a reasonable proxy for replica inconsistency) is well-suited for many streaming applications where having perfect answers at all times is not essential (see Section 2). Our approach also performs well in the face of the non-uniform failure durations observed in empirical measurements of system failures: most failures are short, but most of the downtime of a system component is due to long-duration failures [18, 23].

We implemented our approach in the Borealis distributed stream processing system and present the results of several experiments. We find that Borealis can handle failures of variable duration while ensuring that new tuples are processed within a pre-defined threshold (3 seconds per node in our experiments), even when query networks span multiple processing nodes. We also show that our scheme has low runtime overhead. The main latency overhead comes from SUnion, which needs to buffer tuples temporarily to account for variable propagation delays.

2. MODEL, ASSUMPTIONS, AND GOALS

This section describes our distributed stream processing model, the failure assumptions underlying our work, and our design goals.

2.1 Query and Failure Model

A loop-free, directed graph of operators processing data arriving on streams forms a *query network*. In many stream processing applications, input streams arrive from multiple sources across the network, and are processed by a *Union* operator that produces a FIFO order of the inputs before further processing. These inputs may come directly from data sources, such as network monitors sending synopses of connection or other activity, or may be the results of processing at upstream SPE nodes. Figure 1 illustrates a query network distributed across four SPEs.

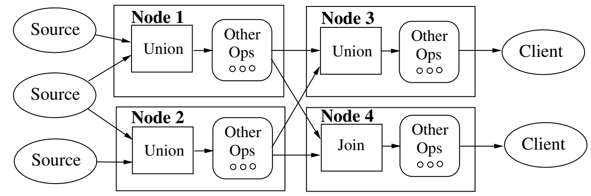

Figure 1: Query network in a distributed SPE.

To avoid blocking in face of infinite input streams, operators process windows of tuples. Some operators, such as *Join* may still block when some of their inputs are missing. In contrast, a Union is an example of a *non-blocking* operator because it can perform meaningful processing even when some of its inputs streams are missing. In Figure 1, the failure of a data source would not prevent the system from processing the remaining streams. Failure of node 1 or 2 would not block node 3 but would block node 4.

When a *partial failure* occurs *upstream* from a non-blocking operator and causes some (but not all) of its input streams to be unavailable, it is often useful to process the inputs that remain available. There are many applications, including real-time network monitoring and sensor network data monitoring, where continuing processing with partially available data is preferable to blocking.

For example, in real-time network monitoring, even if only a subset of monitors are currently available, processing their data might help quickly identify potential attackers or other network anomalies. The information provided by the subset, however, may not lead to accurate conclusions in all cases. For instance, some source IP addresses might just happen to send all their traffic through a subset of monitors and might appear more active than they should just because the total count is lower than the actual value. Hence, it is important to arrange for an SPE to correct its internal state and forward correct results downstream when a failure heals and previously unavailable data streams are made available.

2.2 Failure Assumptions

Our approach handles non-Byzantine fail-stop failures (*e.g.*, software crashes), network failures, and network partitions where any subset of SPE nodes lose connectivity to one another. When each processing node has N replicas (including itself), we tolerate at most $N-1$ simultaneous node failures. If a network connection delays tuples longer than a pre-defined period, we assume that a failure has occurred.

We assume data sources and clients also implement the fault-tolerance protocols described in the next section. This can be achieved by having clients use a fault-tolerant library or by having them communicate with the system through proxies (or nearby processing nodes) that implement the required functionality. These proxies can then perform the failure handling on their client's behalf. For instance, when a node receives data straight from a data source, it can log it persistently (*e.g.*, in a transactional queue [9]) and transmit it to all replicas that should process that stream.

Our scheme is optimized for the case when the SPE nodes change infrequently and are interconnected with relatively high-bandwidth links (tens of Mbits/s or more). Nodes communicate with each other with a reliable, in-order protocol like TCP. The query network at any SPE node is replicated at a small number of other nodes.

2.3 Design Goals

Our goal is to ensure that any data tuple on an input stream is processed within a specified time bound, regardless of whether failures have occurred on other input streams or not. Among the many possible ways to achieve this goal, we seek methods that produce the fewest unstable tuples during a failure. If $\text{Delay}_{\text{new}}$ is the maximum delay before an SPE node processes a new input tuple, and $\text{N}_{\text{unstable}}$ the number of unstable tuples produced by that SPE, our goal is to minimize (for each SPE) $\text{N}_{\text{unstable}}$, subject to $\text{Delay}_{\text{new}} < X$. X is an application- or user-specified time threshold.

Reducing $\text{N}_{\text{unstable}}$ has two benefits. First, it reduces the amount of resources consumed by downstream nodes in processing unstable tuples. Second, $\text{N}_{\text{unstable}}$ may be thought of as a (crude) substitute for the degree of divergence between replicas when the set of input streams is not the same at the replicas. Reducing replica divergence makes it easier and faster to reconcile replica state after a failure heals.

Our approach ensures that as long as some path of processing non-blocking operators is available between one or more data sources and client application, the client will receive results. If multiple such paths are available, our schemes favor stable results over unstable ones, but do not guarantee that. Once failures heal, however, we ensure that the client receives stable versions of all results, and that all the replicas converge to the same state. We handle both single failures and multiple overlapping (in time) failures.

3. APPROACH

This section describes our replication scheme and underlying algorithms. Each SPE implements the state machine shown in Figure 2 that has three states: STABLE, UPSTREAM_FAILURE, and STABILIZING.

As long as all upstream neighbors of an SPE are producing stable tuples, the SPE is in STABLE state. In this state, it processes tuples as they arrive and passes stable results to downstream nodes. To maintain consistency between replicas that may receive inputs in different orders, we define a data-serializing operator, *SUnion*. Section 3.2 discusses the STABLE state and the SUnion operator.

Each SPE node sends periodic heartbeat messages to its downstream nodes. When an SPE finds a broken upstream connection, a missing heartbeat, or receives unstable tuples, it goes into the UPSTREAM_FAILURE state. In that state, the SPE has three options for processing new data:

1. *Suspend* until the failure heals and the failed upstream nodes start producing data again.
2. *Delay* without suspending the processing of each tuple.
3. *Process* each tuple without any delay.

The first option favors consistency for availability. It does not produce any unstable tuples but it may be used only for short failures given our goal to process new tuples with bounded delay. The latter two options both produce result tuples that are marked "unstable"; the difference between the options is in the latency of results and the number of unstable tuples produced. Section 3.3 discusses the UPSTREAM_FAILURE state.

A failure *heals* when a previously unavailable upstream node starts producing stable tuples again or when a node finds another replica upstream that can provide stable tuples. The SPE transitions to the STABILIZING state at this stage (Section 3.4). In this state, the upstream node sends correct (stable) versions of previously missing or unstable tuples for the SPE to process. If the downstream node processed any unstable tuples during UPSTREAM_FAILURE it must reconcile its state and stabilize its outputs. We explore two approaches for state reconciliation: a checkpoint/redo scheme and an undo/redo scheme.

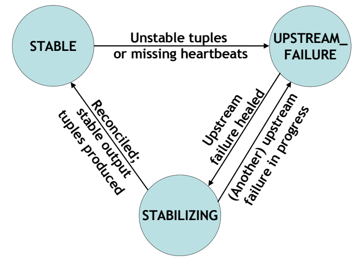

Figure 2: The Borealis state machine.

While reconciling, new input tuples are likely to continue to arrive. The SPE has the same three options mentioned above for processing these tuples: suspend, delay, or process without delay. Processing new tuples while reconciling entails setting up two versions of the SPE's query network: a *time-shifted version* that reconciles state by reprocessing the stable versions of previously missing or unstable tuples, and a *current version* to process newly arriving tuples (and possibly output additional unstable tuples.)

Once the reconciliation is complete, the node transitions to the STABLE state if there are no other current failures, or to the UPSTREAM_FAILURE state otherwise.

3.1 Data Model

With our approach, SPE nodes and client applications must distinguish between results produced based on stable inputs and other results. Furthermore, stable tuples produced after stabilization may override previous unstable ones, requiring a node to correctly process these amendments. To accommodate these new tuple semantics, we adopt and extend the data model introduced in Borealis [2].

A stream is an append-only sequence of tuples of the form: $(k, a_1, \ldots, a_m)$, where k uniquely identifies the tuple in the stream, (*e.g.*, with a timestamp) and $a_1, \ldots, a_m$ are attribute values. As in Borealis, we extend tuples to take the form:

$$(\text{tuple\_type}, \text{tuple\_id}, \text{t\_min}, a_1, \ldots, a_m)$$

In this extended model:

1. **tuple_type** indicates the type of the tuple. Traditionally, all tuples are stable insertions. We introduce two new types of tuples: UNSTABLE and UNDO. An UNSTABLE tuple is one that may subsequently be amended with a stable version. UNDO indicates that *all tuples following the identified one seen thus far on the stream are to be deleted*, and the associated state of any operators rolled back. Stable tuples that follow the UNDO replace the undone unstable tuples. We use a few additional types of tuples in our approach but they do not fundamentally change the data model. Some tuples even only exist on separate control streams that our new operators use to communicate with their SPE. Table 1 summarizes the new tuple types.
2. **tuple_id** uniquely identifies the tuple in the stream.
3. **t_min** is the timestamp of the earliest input tuple that affected the value of this tuple.

With this model, applications that do not tolerate incon-

Tuple type	Description
Data streams	
STABLE	Regular tuple
UNSTABLE	Tuple that may be corrected later
UNDO	All tuples following the one identified should be removed
BOUNDARY	All following tuples will have a timestamp equal or greater to the one indicated
UNDO_START	Control message from runtime to SUnion to trigger undo-based recovery
REC_DONE	Tuple that indicates the end of reconciliation
Control streams	
UNSTABLE	SUnion signals the beginning of a failure
REC_REQUEST	SUnion signals that a failure healed and the state should be reconciled
REC_DONE	SOutput signals the end of reconciliation

Table 1: Types of tuples

sistencies may simply drop all unstable and undo tuples.

3.2 Stable State

A stream processing operator is deterministic if its results do not depend on the times at which its inputs arrive (*i.e.*, the operator does not use timeouts); of course, the results will usually depend on the input data order. Our replication scheme handles query networks composed of deterministic operators. When multiple replicas of an SPE node receive inputs from the same upstream sources, we need a way to ensure that each replica of an operator processes data in the same order; otherwise, the replicas will diverge even in the absence of failures. Ideally, this task should not require any inter-replica communication.

To meet this goal, we propose a simple data-serializing operator, *SUnion*. SUnion takes multiple streams as input and applies a deterministic sort function on *buckets* of tuples. This sort ensures that all tuples in a bucket are processed by the next operator in the same order by all the replicas, assuming that all the replicas have seen the same data tuples (albeit possibly in different orders).

SUnion requires that all data sources, or SPE nodes acting on their behalf, timestamp tuples (*i.e.*, set their t_min) with a synchronized clock and stream them to all replicas. Each replica buffers tuples whose t_min fall in a pre-defined bucket and sorts them once the bucket is full. Within each timestamp-defined bucket, any sort order may be used. The sort may operate on any attribute of the tuples, not just the timestamp, and that attribute may be selected dynamically. We assume that when an SPE is in STABLE state, the difference in tuple arrival times between streams falls within at most a few buckets.

To distinguish between failures and lack of data, data sources send periodic heartbeats in the form of *boundary tuples*. These tuples have tuple_type = BOUNDARY and each data source guarantees while sending a boundary tuple that no tuples with t_min smaller than the boundary's t_min will be sent subsequently. BOUNDARY tuples are similar to punctuation tuples [39] or heartbeats [36]. The time interval between such tuples should be no larger than the size of a bucket to avoid delaying tuples unnecessarily. Missing boundary tuples indicate a failure.

Figure 3 illustrates the serialization of three streams. Tuples in bucket i can be sorted and forwarded as stable because boundary tuples with timestamps greater than the bucket boundary have arrived. Neither of the other buckets can be processed. Both buckets are missing boundary tuples

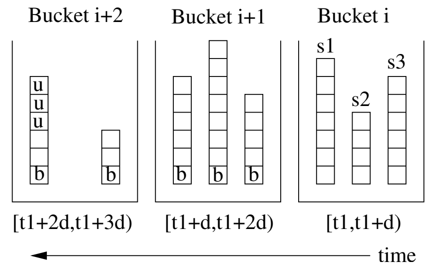

Figure 3: Example of serialization of streams s_1, s_2, and s_3 with boundary interval d. Tuples are grouped into buckets by timestamp values. The u's denote unstable inserts and b's denote boundary tuples.

and bucket $i + 2$ even contains unstable tuples.

Our approach is similar to the Input Manager in STREAM [36]. In contrast to the Input Manager, whose goal is to sort tuples by increasing timestamp order and deduce heartbeats if applications do not provide them, our goal is to ensure that replicas process tuples in the same order and for nodes to distinguish between failures and delays on the input streams. In case of failure or excessive delay, a node should proceed with the remaining tuples. As we discuss in the next section, when the failure heals and the missing tuples become available, the node should reconcile and reprocess its inputs. The Input Manager does not make such distinctions; it assumes that delays are bounded.

Because tuples on streams may not be ordered on any attribute or on boundary timestamps, SUnions placed deeper in the query network must use the same boundary timestamps to organize tuples into buckets and sort them. We thus propagate boundary tuples through the query network.

Tuples entering the system have t_min set by the data sources (or the first processing nodes). When operators produce a tuple, they set the t_min to be the oldest timestamp of all input tuples that contributed to this output tuple. For a join, it is the lower of the two timestamps of the joined tuples. For an aggregate it is the minimum timestamp of all tuples in the window.

3.3 Upstream Failure

When a failure occurs an upstream node may fail, block, become unreachable, or start producing unstable outputs. The downstream node then stops receiving boundary tuples and either stops receiving data or receives unstable data from that neighbor. In both cases, the downstream node enters the UPSTREAM_FAILURE state.

The goal of our approach is to reduce N_{unstable} subject to $\text{Delay}_{\text{new}} < X$. For short failures, a replica should thus block and ensure $N_{\text{unstable}} = 0$ while $\text{Delay}_{\text{new}} = D + P$, where D is the duration of the failure and P is the normal processing delay (including queuing delays).

To avoid producing unstable tuples even for longer failures, we propose that a node in UPSTREAM_FAILURE tries to find an alternate stable upstream replica, within a time smaller than X time units. We denote this time by αX. α must satisfy $\alpha X < X - P$. To enable such replica switching, downstream nodes must monitor *all* upstream replicas not just the one currently sending data. For that, we propose that downstream nodes periodically requests heartbeat responses from all replicas. These responses include the state, stable or unstable, of their output streams. When a failure

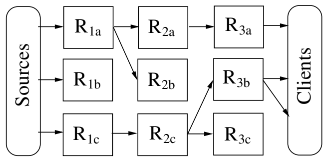

Figure 4: Example of replicated SPEs. R_{ij} **is the** j**'th replica of processing node** i**.**

occurs, the downstream node uses the information from the latest such messages to find an available replica, possibly trying all of them in turn.

Because the new replica should continue sending data from the same point in the stream, stabilizing it first if necessary, when switching to a new replica, a node sends the identifiers of the last stable and unstable tuples it received. We further discuss switching between upstream neighbors in various consistency states in Section 3.5. The result of these switches is that any replica can forward data streams to any downstream replica or client and that the outputs of some replicas may not be used, as illustrated in Figure 4.

If the downstream node fails to find a stable replica within αX time-units, it must unblock new tuples and continue processing them to ensure the required availability level. When this occurs, SUnion operators serialize the available tuples, labeling them as UNSTABLE, and buffering them in preparation for future reconciliation. In the example from Figure 3 if the boundary for stream s_2 does not arrive within αX of the time the first tuple entered bucket $i + 1$ or bucket $i + 2$ still contains unstable tuples αX time units after the first tuple entered that bucket, the remaining tuples in these buckets are stored and forwarded as unstable. As the SPE processes unstable tuples, its state may start to diverge.

As the node processes unstable tuples it can do one of two things: delay new tuples as much as possible or process them without delay. Continuously delaying new tuples helps reduce the number of unstable tuples produced during failure but it constrains what the node can do during stabilization, as we discuss next.

3.4 Stabilization

A failure heals when a node that was missing an input stream is able to communicate again with a stable upstream replica for that stream, and receives a replayed version of the missing inputs. To ensure consistency, the node must reconciles its state to reach a consistent state and stabilizes its outputs; *i.e.*, replace previously unstable tuples with their stable counterparts thus allowing its downstream neighbors to reconcile their states and stabilize their outputs in turn. We present state reconciliation and output stabilization techniques in this section.

In the STABILIZING state, while reconciling its state, a node may need to continue processing new tuples to meet the $\text{Delay}_{\text{new}} < X$ availability requirement. To achieve that, we present an approach based on running two versions of a query network, one handling reconciliation and one processing new data. The two versions can run on a single node or they can be two existing replicas.

3.4.1 State Reconciliation

Because no replica may have the correct state after a failure and because the state of an SPE depends on the exact sequence of tuples it processed, we propose that an SPE node reconciles its state by reverting it to a pre-failure state and reprocessing all input tuples since then. To revert to an earlier state, we explore two approaches: reverting to a checkpointed state or undoing the effects of unstable tuples. Both approaches require that the node suspends processing new input tuples while reconciling its state.

Checkpoint/redo reconciliation. In this approach, the SPE periodically checkpoints the state of the query network when it is in the STABLE state. SUnion operators buffer input tuples between checkpoints and they continue to do so during UPSTREAM_FAILURE. Input tuples must be buffered between checkpoints because they will be replayed if the node restarts from the checkpoint. When a checkpoint occurs, however, SUnion operators truncate all buckets that were processed before that checkpoint.

To perform a checkpoint, the SPE suspends all tuple processing and iterates through all operators and intermediate queues to make a copy of its state. Checkpoints could be optimized to copy only the difference in the state since the last checkpoint thus reducing the CPU overhead when the state changes slowly compared with the checkpoint interval. We do not investigate this optimization in the paper, though.

To reconcile its state, a node re-reads its state from the checkpoint and reprocesses all buffered input tuples. The reconciliation time is thus the time to copy the state and reprocess all tuples that arrived since the last checkpoint before failure.

Undo/redo reconciliation. To avoid the CPU overhead of checkpointing and to recover at a finer granularity by rolling back only the state on paths affected by the failure, another approach is to reconcile by undoing the processing of unstable tuples and redoing that of their stable counterparts. With undo/redo, SUnion operators only need to buffer unstable buckets, truncating stable ones as soon as they process them.

To support such an approach, all operators should implement an "undo" method, where they remove a tuple from their state and, if necessary, bring some tuples previously evicted from the state back into the current window. Supporting undo in operators may not be straightforward—for example, suppose an input tuple, p, caused an aggregate operator to close a window and output a value. To undo p, the aggregate must undo its output but must also bring back all the evicted tuples and reopen the window.

Instead, we propose that operators buffer their input tuples and undo by *rebuilding the state* that existed right before they processed the tuple that must now be undone. To determine how far back in history to restart processing from, operators maintain a *set of stream markers* for each input tuple. The stream markers for a tuple p in operator u are identifiers of the oldest tuples on each input stream that still contribute to the operator's state when u processes p. To undo the effects of processing all tuples following p, u looks up the stream markers for p, scans its input buffer until it finds that bound, and reprocesses its input buffer since then, stopping right after processing p. A stream marker is typically the beginning of the window of tuples to which p belongs. Stream markers do not hold any state. They are rather like pointers to some location in the input buffer.

Operators that keep their state in aggregate form must explicitly remember the first tuple on each input stream that begins the current aggregate computation(s). In the worst case, determining the stream markers may require a linear scan of all tuples in the operator's state. To reduce the runtime overhead, operators may set stream markers periodically at the expense of a slightly longer reconciliation time. The reconciliation times is thus the time spent processing the undo history backwards up to the correct stream markers and reprocessing all tuples since then.

3.4.2 Stabilizing the Output Streams

Independently of the approach chosen by a replica to reconcile its state, a node always stabilizes its output streams by producing a single UNDO tuple followed by the stable versions of all the deleted tuples. UNDO indicates that *all* tuples following the identified one should be removed and their effects rolledback

When they receive an UNDO tuple, SUnion operators at downstream nodes stabilize the input streams by replacing the tuples in their buffers with their stable counterparts and triggering a state reconciliation at the node.

With undo/redo reconciliation, operators process and produce UNDO tuples, which propagate also to downstream nodes. To generate an UNDO tuple with checkpoint/redo, we introduce a new operator, *SOutput* that we place on all output streams. At runtime, SOutput acts as a pass-through filter that also remembers the last stable tuple it produced. During checkpoint recovery, SOutput drops duplicate stable tuples and produces the UNDO tuple.

Once stabilization completes, a node transmits a REC_DONE tuples to its downstream neighbors. Reconciliation finishes when the node re-processes all previously unstable input tuples *and* catches up with normal execution (i.e., it clears its queues) or another failure occurs and the node goes back into UPSTREAM_FAILURE. SOutput operators generate and forward the REC_DONE tuples.

3.4.3 Processing New Tuples During Reconciliation

After long failures, the reconciliation itself may take longer than X. A node then cannot suspend new tuples while reconciling. It must produce both corrected stable tuples and new unstable tuples. We propose to achieve this by using two replicas of a query network: one replica remains in unstable state and continues processing new input tuples while the other replica performs the reconciliation. Both replicas stream their outputs in parallel. Hence, unstable tuples that appear between an UNDO and a REC_DONE correspond to new tuples while stable tuples correct earlier unstable ones. Unstable tuples that appear after the REC_DONE correspond to a new failure. We discuss how to ensure these semantics in spite of failures during recovery in Section 3.5.

Once again, we have a trade-off between availability and consistency. Suspending new tuples during reconciliation reduces the number of unstable tuples but may eventually break the availability requirement. Processing new tuples during reconciliation increase the number of unstable tuples but the SPE may still attempt to reduce their number by delaying new tuples as long as possible. We compare these alternatives in Section 5.1.

To create the second version of the query network, a node can replicate its state in another locally running SPE and run both versions of the query network locally. Because we already use replication, however, we propose that replicas use each other as the two query network versions as follows. When a pair of replicas can hear each other during a failure, they form a partnership. The first partner that needs reconciliation moves its clients to the second partner. The second partner postpones its own reconciliation until the first one finishes. When the first partner completes its reconciliation, it takes both sets of clients back and sends them the corrected output streams in the form of a single UNDO tuple followed by the final correct information. The partner's downstream clients might need to undo from a different position in the stream but this information can easily be exchanged with the list of clients. The second replica can then perform its own reconciliation. In case both partners need to reconcile simultaneously, they can use their node identifiers to break the tie. Replicas establish partnership with randomly chosen other replicas.

3.5 Analysis

In this section, we discuss the main properties of our approach. To help us state these properties, we start with a few definitions.

A data source *contributes to a stream*, s, if it produces a stream that become s after traversing some sequence of operator replicas, called a *path*. The union of paths that connect a set of sources to a destination (*i.e.*, a client or an operator), such that any operator that appears on more than one path corresponds to the same replica of that operator, forms a *tree*. If the set of sources contains all data sources that contribute to the stream, the tree is *stable*, because it produces stable tuples during execution. If any of the missing sources from a tree would connect to it through non-blocking operators, the tree is *unstable*. Otherwise, the tree is *blocking*.

PROPERTY 1. *In a static failure state, if there exists a stable tree, a destination receives stable tuples. If only unstable trees exist, the destination receives unstable tuples from one of the unstable trees. Otherwise, the destination may block.*

The above property comes directly from the ability of downstream nodes to monitor and switch upstream neighbors, preferring stable ones over unstable ones.

PROPERTY 2. *A destination receives the results from input streams reachable through a stable or unstable tree within at most a kX time unit delay, where k is the number of processing nodes on the longest path in the tree.*

Our model is that an application-defined maximum end-to-end delay, kX, gets equally divided across processing nodes. Each node processes new tuples within X time units. We examine this property further in Section 5.

PROPERTY 3. *Switching between trees never causes duplicate tuples and may only lose unstable tuples.*

We discuss the above property by examining each possible upstream-neighbor switching scenario. In our approach, every node buffers its input and output tuples. We assume that these buffers can hold more tuples than the maximum number that can be delivered during a single failure and recovery. In case of excessively long failures, buffers could be written to disk. To trim these buffers, node could use queue trimming methods as in [25].

1. *Switching between stable upstream neighbors*: Because the downstream node indicates the identifier of the last stable tuple it received, a new stable replica can continue from that point in the stream either by waiting to produce that tuple or replaying its output buffer.

2. *Switching from an unstable to a stable upstream neighbor*: Because the downstream node indicates the identifiers of the last stable and unstable tuples it received, the new neighbor can stabilize the stream and then continue with stable tuples.
3. *Switching to an unstable upstream neighbor*: This is the only scenario where a node must drop unstable tuples. In this scenario, the new neighbor may be in a completely different state than the previous one and it may have been unstable longer. Because nodes cannot undo stable tuples, the new upstream and downstream pair may have to continue processing tuples while in mutually inconsistent states. To avoid duplicating any results even partially, we add a second timestamp, t_max to tuples. t_max of a tuple p is the timestamp of the *most recent* input tuple that affected p. To avoid duplications, the new upstream node forwards only output tuples that have a t_min greater than the highest t_max that the downstream node previously received. This ensures that the new tuples result from processing a non-overlapping sequence of tuples but may result in unstable tuple loss for the downstream node.

PROPERTY 4. *As long as one replica of each processing node never fails, when all failures heal, the destination receives the complete and corrected stable stream.*

When all failures heal, the processing nodes that receive their inputs directly from sources, receive if necessary, the replay of the complete correct input. They reconcile their states and stabilize their outputs. Their downstream neighbors can then reconcile in turn. The procedure continues until all nodes reconcile their state. Each node corrects all unstable tuples it produced, so the destination eventually receives the correct stable stream.

PROPERTY 5. *Stable tuples are never undone.*

We now show that our approach correctly handles failures during failures without the risk of undoing stable tuples. Let's assume that a node starts reconciling its state using undo/redo. SUnion produces an undo tuple followed by the stable version of all tuples processed during the failure. If new unstable tuples come from upstream they are processed afterward and the new failure follows the reconciliation without affecting it. Each operator will undo and redo its state before seeing the new unstable tuples, hence no previously stable tuple will need undoing.

Now let's assume that while the undo tuple propagates, a different input stream becomes unstable and both streams merge at one operator. Independently of which tuples arrive at the operator first, when the operator finally processes the UNDO tuple, it rebuilds the state it had *before the first failure* and processes all tuples that it processed during that failure *before* starting to process any new unstable tuples. The operator thus produces an UNDO tuple followed by stable tuples that correct the first failure, followed by the unstable tuples from the new failure. Once again, the new failure appears to occur after stabilization.

If a node uses checkpoint redo, we assume it reverts the whole query network to a pre-failure state and starts processing tuples since then. To avoid undoing any stable tuples, the runtime must force a checkpoint between the time SOutput operators produced their UNDO tuples and the time any of the SUnion operators push new unstable tuples into the query network. This will be a snapshot of the reconciled

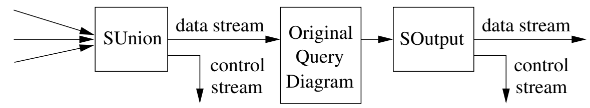

Figure 5: Modified query network.

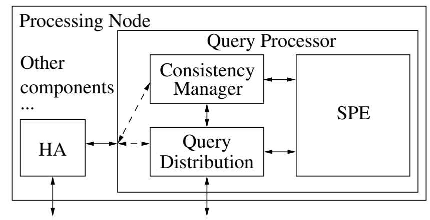

Figure 6: Extended software node architecture.

state and the node will restart from this new checkpoint after the second failure heals. To achieve this, SUnion operators must simply delay serializing new unstable tuples until they have seen a checkpoint.

Hence, in all cases when a failure occurs before the end of reconciliation, the downstream node sees an UNDO tuple followed by stable tuples correcting those from the first failure and followed by unstable tuples from the new failure. SOutput produces a REC_DONE tuple between the last stable and first unstable tuples that it sees.

4. IMPLEMENTATION

In this section, we present the implementation of our approach in Borealis. In addition to inserting *SUnion* and *SOutput* operators into the query network, we add a *Consistency Manager* and an *HA* ("high availability") component to each SPE node. We assume nodes already have components that manage query distribution. Figure 5 and 6 illustrate these modifications (arrows indicate communication between components). We also require operators to implement a simple API.

The HA component monitors all replicas of a node and all its upstream neighbors (and their replicas). It informs the query processor of changes in their states.

The Consistency Manager makes all decisions related to failure handling. In STABLE state, it periodically requests that the SPE checkpoints the state of the query network. When the node must reconcile its state, the Consistency Manager chooses whether to use undo/redo or checkpoint/redo and when to ask a partner to produce new tuples. In case of undo/redo, the Consistency Manager injects an UNDO_START tuple on one the affected SUnion input streams. For checkpoint redo, the Consistency Manager requests that the SPE performs checkpoint recovery.

In addition to their tasks described in previous sections, SUnion and SOutput communicate with the Consistency Manager through extra *control* output streams. When an SUnion can no longer delay tuples, it informs the Consistency Manager about the UPSTREAM_FAILURE, by producing an UNSTABLE tuple on its control stream. Similarly, when input streams are corrected and the node can reconcile its state, SUnion produces a REC_REQUEST tuple. Once reconciliation finishes, SOutput forwards a REC_DONE tuple on its control stream.

Borealis requires the following modifications to existing operators. For checkpoint/redo, operators need the ability to take snapshots and recover their state ((un)packState methods). For undo/redo, each operator must periodically compute its stream markers and must re-read tuples from its input queue when an undo tuples arrives. This functionality can be implemented with a wrapper, requiring that the operator itself only implements two methods: clear() clears the operator's state and findOldestTuple(int stream_id) returns the oldest tuple from input stream, stream_id, that is currently in the operator's state. To propagate boundary tuples, operators must implement findOldestTimestamp() method that returns the oldest timestamp in the operator's state. This value is the smaller of the oldest timestamp present in the operator's state and the oldest timestamp in the boundary tuples received on all input streams.

In our approach, nodes communicate with each other through the data they exchange but also through separate control messages that enable the nodes to rebuild failed data paths and handle each other's clients during reconciliation. To modify the data path, nodes send each other subscribe and unsubscribe messages with either their own information if they need to receive a stream or information about their clients if they would like the other node to handle these clients temporarily during reconciliation. We assume that every node has enough bandwidth to handle a large subset of all downstream clients and replicas. If this is not the case, a node may have to distribute its clients across multiple other replicas during reconciliation.

When a node fails and recovers, we assume it restarts its query diagram fragment from an empty state and refuses new clients until it processes sufficiently many tuple to reach a consistent state.

5. EVALUATION

In this section, we evaluate the performance of the approach through experiments with our prototype implementation. All experiments were performed on a 3 GHz machine with 2 GB of memory running Linux (Fedora Core 2).

We first examine the performance of a single Borealis node in the face of temporary failures of its input streams. In particular, we compare in terms of $\mathrm{Delay}_{\mathrm{new}}$ and $\mathrm{N}_{\mathrm{unstable}}$ different strategies regarding suspending, delaying, and processing new tuples during upstream failure and reconciliation. In these experiments the node uses checkpoint/redo to reconcile its state. Second, we examine the performance of our approach when failures and reconciliation propagate through a sequence of processing nodes. Third, we compare the undo/redo and checkpoint/redo reconciliation techniques. We finally discuss the overhead of our approach.

In our prototype, it takes a node approximately 40 ms to switch between upstream neighbors. Given that this value is small compared with αX, our system masks node failures within the required availability constraints. We thus focus the evaluation on failures of input streams.

5.1 Single-Node Performance

The optimal approach to handling failures shorter than αX, is to delay processing new tuples until the failure heals. This minimizes $\mathrm{N}_{\mathrm{unstable}}$ without breaking the constraint on $\mathrm{Delay}_{\mathrm{new}}$ (we use $\alpha = 0.9$ in all experiments). When a failure exceeds αX, a processing node can either continuously delay new tuples by αX or catch-up and process new tuples almost as they arrive. We call these alternatives Process and Delay and examine their impact on $\mathrm{Delay}_{\mathrm{new}}$ and $\mathrm{N}_{\mathrm{unstable}}$.

We run a query network composed of an SUnion, a Join, and SOutput with three input streams and an aggregate rate of 3000 tuples/s. The join serves as a generic query network with a 100 tuple state size thus creating a light load. We cause a 5 s failure during which we continue producing tuples but do not send them on one of the streams. After the failure, we send all missing tuples while continuing to stream new tuples. We vary αX from 500 ms to 6 s and observe $\mathrm{Delay}_{\mathrm{new}}$ and $\mathrm{N}_{\mathrm{unstable}}$ until after stabilization completes for both Process and Delay. Figure 7 shows the results.

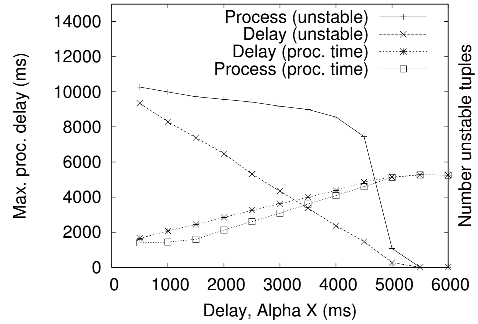

Figure 7: Delaying tuples during UPSTREAM_FAILURE reduces $\mathrm{N}_{\mathrm{unstable}}$. Y-Axes: left $\mathrm{Delay}_{\mathrm{new}}$, right $\mathrm{N}_{\mathrm{unstable}}$. Failure duration: 5 s.

The techniques differ significantly in the number of unstable tuples they produce. With Process, as soon as the initial delay is small compared with the failure duration ($\alpha X \leq 4$ *s* for a 5 s failure), the node has time to catch-up and produces a number of unstable tuples almost proportional to the failure duration. The $\mathrm{N}_{\mathrm{unstable}}$ graph approximates a step function. In contrast, Delay reduces the number of unstable tuples proportionally to αX. With both approaches, $\mathrm{Delay}_{\mathrm{new}}$ increases linearly with αX. It is slightly lower for Process because the first *new* tuple after reconciliation has been in the queue for a shorter time but also appears later in that queue. From the perspective of our optimization, Delay is thus better than Process for upstream failures.

Delaying new tuples as much as possible during UPSTREAM_FAILURE leads to fewer unstable tuples. When the failure heals, however, the node may be forced to continue processing new tuples because there is no room left for added delays. The question is whether it is ever better to process tuples faster during upstream failure in order to have more slack to suspend producing new tuples completely or at least continue delaying new tuples during stabilization. This is particularly interesting because failures cause nodes to process fewer input tuples, thus possibly producing fewer output tuples in UPSTREAM_FAILURE than during STABILIZATION.

To answer the above question, we examine all six combinations of processing (Process) and delaying (Delay) new tuples during failure and later either suspending new tuples while reconciling the state (Suspend) or having a second version of the SPE continue processing them with or without delay (Delay or Process). Figure 8 shows $\mathrm{Delay}_{\mathrm{new}}$ and $\mathrm{N}_{\mathrm{unstable}}$ for each combination and for increasing failure durations. Figure 9 shows the same results for longer failures. We only show results for failures up to 1 minute. Longer experiments were continuing the same trends. We use the same experimental setup as above but with 4500 tuples/s to emphasize differences between approaches and with $X = 3$ s. Each point is an average of four experiments.

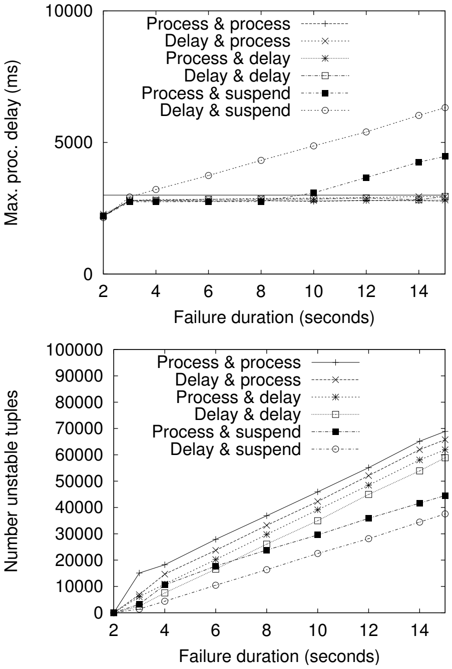

Figure 8: $\text{Delay}_{\text{new}}$ (top) and $\text{N}_{\text{unstable}}$ (bottom) for each combination of delaying, processing, and suspending during UPSTREAM_FAILURE and STABILIZATION. Each approach offers a different consistency-availability trade-off. X-axis starts at 2 s.

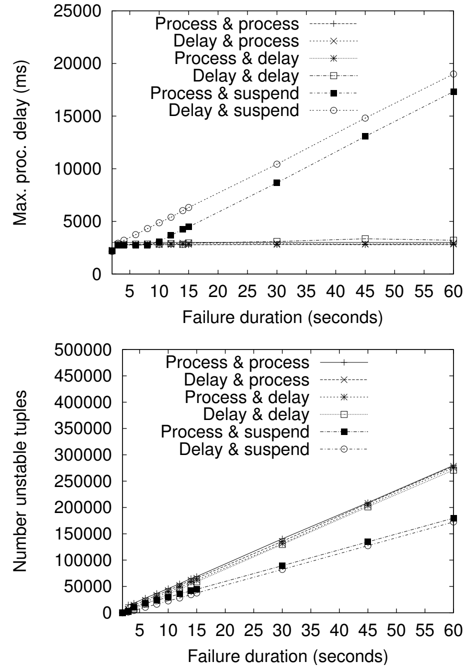

Figure 9: Same experiment as Figure 8 but for failures up to 1 minute.

Because blocking is optimal for short failures, all approaches block for $\alpha X = 2.7$ s and produce no unstable tuples for failures below that threshold. Delaying tuples in UPSTREAM_FAILURE and suspending them during STABILIZATION (Delay & Suspend) is unusable for failures longer than 3 s because it breaks the $\text{Delay}_{\text{new}} < X$ requirement as reconciliation last longer than 300 ms. (Figure 8(top)). This combination is therefore of no interest.

Continuously processing new tuples during both upstream failure and stabilization (Process & Process) ensures that the maximum delay always remains below αX independently of failure duration but produces the most unstable tuples as it produces them for the duration of the whole failure and reconciliation. We can save some unstable tuples by delaying new tuples during stabilization (Process & Delay) without hurting $\text{Delay}_{\text{new}}$. We can also save some unstable tuples by delaying new tuples during upstream failure (Delay & Process). The savings is less than with Process & Delay, however. Indeed, when stabilization starts and we switch from delaying new tuples to processing them, the second replica quickly catches up and undoes most effects of the initial delay. We save most unstable tuples by delaying new tuples both during failure and reconciliation (Delay & Delay). The problem with this approach, however, is that the node is continuously on the verge of breaking the availability guarantee and any burst on input streams may cause it to do so. As shown in Figure 9(top), for the 45 s failure, because we stream all corrected tuples in one burst, $\text{Delay}_{\text{new}}$ slightly exceeds the pre-defined threshold.

More importantly, as D becomes large compared with X (Figure 9), the difference between delaying and processing becomes insignificant compared with the total number of unstable tuples. Since continuously processing unstable tuples reduces latency during the failure and leaves slack for the node to absorb larger bursts in its input streams, it is the most appropriate approach for long failures.

For short failures, however, Process & Suspend may win even over Delay & Delay. If reconciliation is longer than αX (for $D > 6$ s in the experiment), Process & Suspend produces fewer unstable tuples. It is thus better for such failures to process tuples during the failure in order to suspend new tuples during reconciliation. Once reconciliation becomes longer than X, though (for $D > 9$ s), Process & Suspend causes $\text{Delay}_{\text{new}}$ to exceed X. Hence Process & Suspend outperforms Delay & Delay only for failures between 6 and 9 s. Therefore, there exists only a small window when it is better to process new tuples quickly during failures in order to be able to suspend new tuples during reconciliation.

These experiment shows that nodes should handle failures as follows. In UPSTREAM_FAILURE, first delay new tuples as long as possible. As the failure duration increases, process tuples faster. When stabilizing after a short failure, attempt to first suspend new tuples but if reconciliation takes too long, continue processing new tuples with a delay.

5.2 Multiple Nodes

We now examine the performance of our approach in a distributed SPE. We run a network of 1 to 4 SPEs placed in series. Each SPE has one replica and all nodes run the same query network composed of an SUnion, a Join, and an SOutput. Because we run multiple nodes, we reduce the input rate to 1000 tuples/s. We set $X = 2$ seconds, and we cause a 20 second failure on one of the three input streams. It takes about 2 s for the first node to recover. Nodes reconcile their states one after the other.

Figure 10(top) shows the maximum end-to-end processing delay for new tuples. For Process & Process, $\text{Delay}_{\text{new}}$ is

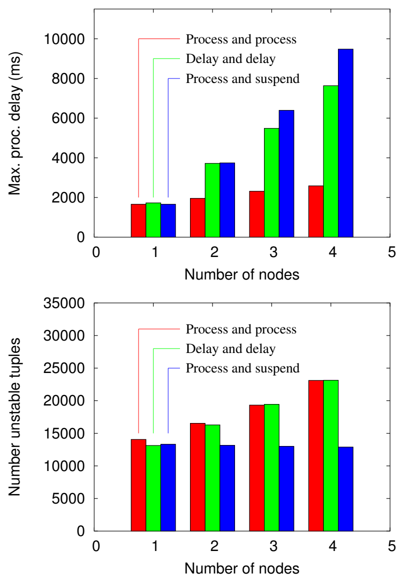

Figure 10: Effects of path length on $\text{Delay}_{\text{new}}$ and $\text{N}_{\text{unstable}}$. Delay or Suspend decrease availability with path length. Delay during UPSTREAM_FAILURE does not reduce $\text{N}_{\text{unstable}}$ for sufficiently long paths.

lowest at αX plus the processing delay through the sequence of nodes. For Delay & Delay, $\text{Delay}_{\text{new}}$ increases by αX for each node in the sequence. Both Process & Process and Delay & Delay keep the end-to-end delay within kX, where k is the number of nodes. For Process & Suspend, $\text{Delay}_{\text{new}}$ is the sum of the stabilization delays. This delay increases for each consecutive node because it has to undo and redo more tuples than its upstream neighbor. Thus, the 3rd and 4th nodes do not meet the $\text{Delay}_{\text{new}} < X$ requirement.

Figure 10(bottom) shows $\text{N}_{\text{unstable}}$ received by the client application. Process & Suspend produces unstable tuples only during the failure. Tuples in the pipeline when the failure heals are never output as nodes reconcile their state instead (in our prototype nodes process tuples only when new tuples arrive). With Process & Process, $\text{N}_{\text{unstable}}$ increases with the length of the chain because all nodes produce unstable tuples while they stabilize and they stabilize one after the other. Interestingly, the savings offered by Delay & Delay decrease with the length of the chain. When stabilization starts, each consecutive node in the sequence runs behind by αX more than its upstream neighbor. When that neighbor stabilizes, both downstream replicas receive *all* tuples until the most recent ones. Because the replica that continues processing new tuples is only supposed to delay new tuples by αX, it catches up and it does so while processing 50% more tuples than the savings during UPSTREAM_FAILURE because all three streams are up and running after the failure.

Overall, for a chain of nodes, Process & Process is clearly the best approach as it maximize availability without paying much penalty in unstable tuples.

5.3 Reconciliation

We now compare the overhead and performance of checkpoint/redo and undo/redo reconciliation. Table 2 summarizes the results. P is the per-node processing delay. p_{comp} is the time to read and compare a tuple. p_{copy} is the time to copy a tuple, and p_{proc} is per-operator processing time.

Figure 11 (left and middle) shows the maximum delay imposed on new tuples during reconciliation, which effectively measures the reconciliation time as we increase the state size, S, of the query network or the number of tuples to reprocess (*i.e.*, $D\lambda$, where D is the failure duration and λ is the aggregate rate on *all* input and intermediate streams). In this experiment, D is 5 seconds and we vary λ. For both approaches, the time to reconcile increases linearly with S and $D\lambda$. When we vary the state size or tuple rate, we keep the other parameter low at 1000 tuples/s and 20 tuples respectively. Each point is the average of three experiments as experiments produced almost identical values.

Undo/redo takes longer to reconcile primarily because it must rebuild the state of the query network ($S(p_{\text{comp}} + p_{\text{proc}})$) rather than recopy it (Sp_{copy}), as shown in Figure 11(left). Interestingly, even when we keep the state size small and vary the number of tuples to reprocess (Figure 11(middle)), checkpoint beats undo/redo, while we would expect the approaches to perform the same ($\propto (D + 0.5l)\lambda p_{\text{proc}}$). The difference is not due to the undo history because when we do not buffer any unstable tuples in the undo buffer (Undo "limited history" curve), the difference remains. In fact, an SPE always blocks for αX (1 s in this experiment) before going into UPSTREAM_FAILURE. Because we continue checkpointing every 200 ms during that time, we effectively always checkpoint the pre-failure state and avoid reprocessing on average $0.5l\lambda$ tuples, which corresponds to tuples that accumulate between the checkpoint and the beginning of the failure. Undo/redo always pays this penalty, as stream markers are computed only when the join processes new tuples. Splitting the state across two operators in series, simply doubles λ.

Checkpoint/redo achieves faster reconciliation but at the cost of a higher CPU overhead at runtime. Figure 11(right) shows that the time our prototype takes to compute a set of stream markers (*i.e.*, scan the state of one operator) is significantly lower than the time to copy the state of one operator, which is lower than the time to actually perform a checkpoint in a query network composed of a union and a join with increasing state size (*i.e.*, window size). Checkpoints include the overhead of checkpointing all operator's input and output queues. Results are medians of 300 samples. Undo/redo does not incur any CPU overhead for buffering tuples as we simply leave them in the operator's input queues.

The memory overhead for checkpoint/redo is the state size plus the input tuples that accumulate between checkpoints ($S + l\lambda_{in}$, where λ_{in} is the aggregate *input* rate). The memory overhead for undo/redo is higher. Assuming, for simplicity, that the number of tuples to read in order to rebuild the state is equal to the state size, S, the overhead is proportional to the *failure duration*, D, the state size S, the interval between stream marker computations, l, and $\lambda_{\text{stateful}}$, the aggregate rate on all input and intermediate streams that feed stateful operators.

Checkpoint/redo thus appears superior to undo/redo. The main advantage of the undo-based approach, however, is the flexibility to undo any suffix of the input streams and propagate reconciliation only on paths affected by failures.

5.4 Runtime Overhead

In Borealis , in addition to the undo and checkpoint overheads discussed above, the main cause of latency overhead is the SUnion operator. If the sorting function requires the op-

Approach	$\text{Delay}_{\text{new}}$	CPU Overhead	Memory Overhead
Checkpoint	$P + Sp_{\text{copy}} + (D + 0.5l)\lambda p_{\text{proc}}$	$\frac{Sp_{\text{copy}}}{l}$	$S + l\lambda_{in}$
Undo	$P + S(p_{\text{comp}} + p_{\text{proc}}) + (D + 0.5l)\lambda(p_{\text{comp}} + p_{\text{proc}})$	$\frac{Sp_{\text{comp}}}{l}$	$S + (l + D)\lambda_{\text{stateful}}$

Table 2: Performance and overhead of checkpoint/redo and undo/redo reconciliations.

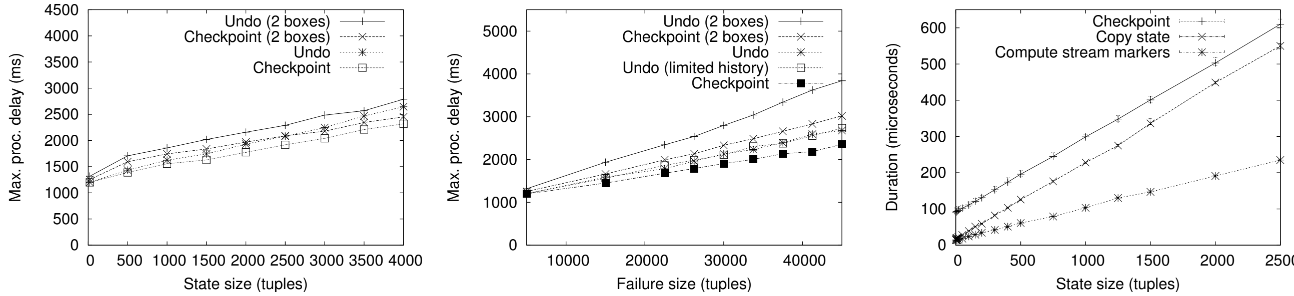

Figure 11: Performance and overhead of checkpoint/redo and undo/redo reconciliations. $\text{Delay}_{\text{new}}$ **for increasing state size (left).** $\text{Delay}_{\text{new}}$ **for increasing failure size (middle). CPU Overhead (right). Checkpoint/redo is faster than undo/redo but has higher overhead.**

Punctuation interval (ms)	50	100	150	200	250	300
Average processing delay	69	120	174	234	298	327
Stddev of the averages	0.5	4	10	28	55	70

Table 3: Latency overhead of serialization.

erator to wait until a bucket is fully bounded before proceeding to the next one, the processing delay increases linearly with bucket size (*i.e.*, boundary interval). Table 3 shows the average end-to-end delay for tuples processed by one node running an SUnion and a Join operator with window size 20 and aggregate input rate 3000 tuples/s (averages of nine 20 second experiments).

With our approach, operators must also check tuple types but this overhead is negligible. They must process boundary tuples for an overhead equivalent to that of computing stream markers. SOutput must also save the last stable tuple that it sees in every burst of tuples that it processes.

6. RELATED WORK

Until now, work on high availability in stream processing systems has focused on fail-stop failures of processing nodes [25, 35]. These approaches have limited tolerance to network failures or partitions—when such failures occur, they either block awaiting data or drop out-of-order data indiscriminately. Other techniques use punctuation [39, 40], heartbeats [36], or statically defined slack [1] to tolerate bounded disorder and delays on input streams. These approaches, however, also block or drop tuples when disorder or delay exceed expected bounds.

Traditional query processing also addresses trade-offs between result speed and consistency, materializing query outputs one row or even one cell at the time [31, 34]. In contrast to these schemes, our approach supports possibly infinite data streams and ensures that once failures heal all replicas produce the same final output streams in the same order.

Fault-tolerance through replication is widely studied and it is well known that it is not possible to provide both consistency and high availability in the presence of network partitions [10]. Eager replication favors consistency by having a majority of replicas perform every update as part of a single transaction [19, 21] but it forces minority partitions to block. With lazy replication all replicas process possibly conflicting updates even when disconnected and must later reconcile their state. They typically do so by applying system- or user-defined reconciliation rules [27, 41], such as preserving only the most recent version of a record [22]. It is unclear how one could define such rules for an SPE. We could copy the state of one replica onto that of another but such reconciliation would be expensive because the state of any node is large and rapidly changing. Additionally, SPEs must also reconcile their input and output tuples to ensure they process the same sequence of tuples after reconciliation.

Other replication approaches use tentative transactions during partitions and reprocess transactions possibly in a different order during reconciliation [22, 38]. With these approaches, all replicas eventually have the same state and that state corresponds to a single-node serializable execution. Replicas notify clients if the output of their tentative transaction changes. Our approach applies the ideas of tentative (unstable) data to stream processing.

Some schemes offer users fine-grained control over the trade-off between precision (or consistency) of query results and performance (*i.e.*, communication resource utilization) [32, 33]. In contrast, we explore consistency/availability trade-offs in face of failures and produce unstable (possibly incorrect) results followed by stable ones.

Workflow management systems [4, 3, 24] commit the results of each execution step (or messages these steps exchange) and use forward recovery in case of failures. The storage servers themselves use one of the standard HA approaches: hot standby, cold standby, 1-safe, or 2-safe [26]. These approaches, however, rely on the assumption that each execution step processes its messages in isolation. This assumption does not hold in a streaming system.

Workflows and systems that process long transactions [20, 43] use compensation to undo the effects of aborted subtransactions. The difficulty in adapting a similar model for streams is that tuples represent data rather than operations. It is unclear how to compensate the effects of a data tuples other than with an undo and redo.

Approaches that reconcile the state after a failure using combinations of checkpoints, undo, and redo are well known [17, 22, 23, 29, 38]. We adapt and use these techniques in the context of fault-tolerance and state reconciliation in an SPE and comparatively evaluate their overhead and performance in these environments.

7. CONCLUSION

In this paper, we presented an approach to fault-tolerant stream processing that handles node failures, network failures, and network partitions. Our approach is based on replication and uses a new data model that distinguishes between unstable tuples, which result from processing partial inputs and may later be corrected, and stable tuples that are immutable. When failures occur, our approach favors availability and produces unstable tuples that it later replaces with their stable counterparts. While ensuring that each node processes new tuples within a pre-defined delay, X, our approach favors failure-handling techniques that reduce the number of unstable tuples.

To ensure consistency at runtime, we introduce a data-serializing operator called SUnion. To ensure consistency after failures heal, nodes reconcile their states using either checkpoint/redo or undo/redo. We find that checkpoints lead to a faster reconciliation but undo can help a node limit the scope of reconciliation to paths affected by the failure.

We implemented the approach in Borealis and showed results from experiments with our prototype. We find that for failures shorter than X time units, to minimize unstable tuples, SPE nodes should block while looking for an available stable upstream replica. As failures become longer, nodes should first delay new tuples as much as possible but then quickly transition to processing tuples as they arrive. Once failures heal, replicas should take turns reconciling their state to ensure that downstream nodes continuously receive the results of processing the most recent data.

8. REFERENCES

[1] Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), Sept. 2003.

[2] Abadi et al. The design of the Borealis stream processing engine. In *CIDR*, Jan. 2005.

[3] G. Alonso and C. Mohan. WFMS: The next generation of distributed processing tools. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[4] Alonso et al. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In *Proc. of IFIP WG8.1 Working Conf. on Information Systems for Decentralized Organizations*, Aug. 1995.

[5] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford University, Oct. 2003.

[6] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, May 2000.

[7] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, June 2003.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, June 2002.

[9] P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *SIGMOD*, June 1990.

[10] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[11] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *29th VLDB*, Sept. 2003.

[12] S. Chandrasekaran and M. J. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *30th VLDB*, Sept. 2004.

[13] Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, Jan. 2003.

[14] Cherniack et al. Scalable distributed stream processing. In *CIDR*, Jan. 2003.

[15] C. Cranor, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: A stream database for network applications. In *SIGMOD*, June 2003.

[16] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, June 2003.

[17] 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.

[18] N. Feamster, D. G. Andersen, H. Balakrishnan, and F. Kaashoek. Measuring the Effects of Internet Path Faults on Reactive Routing. In *ACM Sigmetrics - Performance 2003*, June 2003.

[19] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841 – 860, Oct. 1985.

[20] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, 1987.

[21] D. K. Gifford. Weighted voting for replicated data. In *7th SOSP*, Dec. 1979.

[22] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, June 1996.

[23] J. Gray and A. Reuters. *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.

[24] M. Hsu. Special issue on workflow systems. *IEEE Data Eng. Bulletin*, 18(1), Mar. 1995.

[25] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *21st ICDE*, Apr. 2005.

[26] M. Kamath, G. Alonso, R. Guenthor, and C. Mohan. Providing high availability in very large workflow management systems. In *5th Int. Conf. on Extending Database Technology*, Mar. 1996.

[27] Kawell et al. Replicated document management in a group communication system. In *Second CSCW*, Sept. 1988.

[28] Y.-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *30th VLDB*, Sept. 2004.

[29] D. Lomet and M. Tuttle. A theory of redo recovery. In *SIGMOD*, June 2003.

[30] Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, Jan. 2003.

[31] Naughton et al. The Niagara Internet query system. *IEEE Data Eng. Bulletin*, 24(2), June 2001.

[32] C. Olston. *Approximate Replication*. PhD thesis, Stanford University, 2003.

[33] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, June 2003.

[34] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *SIGMOD*, June 2002.

[35] M. Shah, J. Hellerstein, and E. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *SIGMOD*, June 2004.

[36] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *23rd PODS*, June 2004.

[37] N. Tatbul, U.Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *29th VLDB*, Sept. 2003.

[38] Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th SOSP*, Dec. 1995.

[39] P. A. Tucker and D. Maier. Dealing with disorder. In *MPDS*, June 2003.

[40] P. A. Tucker, D. Maier, and T. Sheard. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Eng. Bulletin*, 26(1), Mar. 2003.

[41] R. Urbano. *Oracle Streams Replication Administrator's Guide, 10g Release 1 (10.1)*. Oracle Corporation, Dec. 2003.

[42] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, June 2002.

[43] H. Wächter and A. Reuter. The ConTract model. In A. K. Elmagarmid, editor, *Database transaction models for advanced applications*. Morgan Kaufmann, 1992.