Fault-Tolerance in the Borealis Distributed Stream Processing System

Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael 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

We present a replication-based approach to fault-tolerant distributed stream processing in the face of node failures, network failures, and network partitions. Our approach aims to reduce the degree of inconsistency in the system while guaranteeing that available inputs capable of being processed are processed within a specified time threshold. This threshold allows a user to trade availability for consistency: a larger time threshold decreases availability but limits inconsistency, while a smaller threshold increases availability but produces more inconsistent results based on partial data. In addition, when failures heal, our scheme corrects previously produced results, ensuring eventual consistency.

Our scheme uses a data-serializing operator to ensure that all replicas process data in the same order, and thus remain consistent in the absence of failures. To regain consistency after a failure heals, we experimentally compare approaches based on checkpoint/redo and undo/redo techniques and illustrate the performance trade-offs between these schemes.

1. INTRODUCTION

In recent years, a new class of data-intensive applications requiring near 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 operators (*e.g.*, filters, aggregates, and correlations) that perform their computations on windows of data that move with time. Most stream processing applications require results to be continually produced at low latency, even in the face of high and variable input data rates. As has been widely noted [1, 9, 14], traditional data base management systems (DBMSs) based on the "store-then-process" model are inadequate for such high-rate, low-latency stream processing.

Stream processing engines (SPEs) (also known as data stream managers [1, 30] or continuous query processors [14]) 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, 6, 16, 28, 41], efficient processing [7, 8, 12, 30], and resource management [13, 17, 30, 35, 38] for SPEs. Stream processing applications are inherently distributed, both because input streams often arrive from geographically distributed data sources, and because running SPEs on multiple *processing nodes* enables better performance under high load [15, 35]. In a distributed SPE, each node produces result streams that are either sent to applications or to other nodes for additional processing. When a stream goes from one node to another, the nodes are called *upstream and downstream neighbors*.

In this paper, we add to the body of work on SPEs by addressing *fault-tolerant stream processing*, presenting a fault-tolerance protocol, implementation details, and experiments. Our approach enables a distributed SPE to cope with a variety of network and system failures. It differs from previous work on high availability in streaming systems by offering a configurable trade-off between availability and consistency. Previous schemes either do not address network failures [25] or strictly favor consistency over availability, by requiring at least one fully connected copy of the query network to exist to continue processing at any time [35]. As such, our scheme is particularly well-suited for applications where it is possible to make significant progress even when some of the inputs are unavailable.

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. In our approach, when a node stops receiving data (or "heartbeat" messages signifying liveness) from one of its upstream neighbors, it requests the missing input streams from a replica of that neighbor (if it can find one). For a node to be able to correctly continue processing after such a switch, all replicas of the same processing node must 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 node is unable to find a new upstream neighbor for an input stream, it must decide whether to continue processing with the remaining (partial) inputs,

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.

or block until the failure heals. If it chooses to continue, a number of possibly incorrect results will be produced, while blocking makes the system unavailable.

Our approach gives the user explicit control of trade-offs between consistency and availability in the face of network failures [11, 22]. We also ensure *eventual consistency*: *i.e.*, clients eventually see the complete correct results. We introduce an enhanced streaming data model in which results based on partial inputs are marked as *tentative*, with the understanding that they may subsequently be modified; all other results are considered *stable* and immutable.

To provide high availability, each SPE processes input data and forwards results within a user-specified time threshold of arrival, even if other inputs are currently unavailable. At the same time, to prevent downstream nodes from unnecessarily having to react to tentative data, an SPE tries to avoid or limit the number of tentative tuples it produces.

When a failure heals, each SPE that processed tentative data reconciles its state by re-running its computation on the correct input streams. While correcting its internal state, the replica also *stabilizes* its output by replacing the previously tentative output with stable data tuples, allowing downstream neighbors to reconcile in turn. We argue that traditional approaches to record reconciliation [27, 42] are ill-suited for streaming systems, and adapt two approaches similar to known checkpoint/redo and undo/redo schemes [18, 23, 22, 29, 39] to allow SPEs to reconcile their states.

Our fault-tolerance protocol addresses the problem of minimizing the number of tentative tuples while guaranteeing that the results corresponding to any new tuple are sent downstream within a specified time threshold. The ability to trade availability (via a user-specified threshold) for consistency (measured by the number of tentative result tuples, since that is often a reasonable proxy for replica inconsistency) is useful in 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 [19, 23].

We have implemented our approach in Borealis [2]. Through experiments, we show that Borealis meets the required availability/consistency trade-offs for failures of variable duration, even when query networks span multiple nodes. We show that it is necessary to process new tuples both during failure and reconciliation to meet the availability requirement for long failures. We find that reconciliation based on checkpoint/redo outperforms reconciliation based on undo/redo because it incurs lower overhead and achieves faster recovery.

2. MODEL, ASSUMPTIONS, AND GOALS

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

2.1 Query and Failure Model

A loop-free, directed graph of operators that process data arriving on streams forms a *query network*. Figure 1 illustrates a query network distributed across four nodes. 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 information or other activity, or may be the results of processing at upstream SPE nodes.

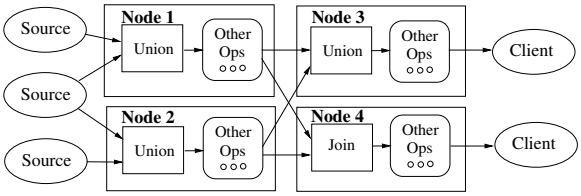

Figure 1: Query network in a distributed SPE.

To avoid blocking in face of infinite input streams, operators perform their computations over windows of tuples. Some operators, such as *Join*, 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 input streams are missing. In Figure 1, the failure of a data source does not prevent the system from processing the remaining streams. Failure of node 1 or 2 does not block node 3 but blocks node 4.

Because many stream processing applications are geared toward monitoring tasks, when a 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 continue processing the inputs that remain available. For example, in a network monitoring application, even if only a subset of monitors are available, processing their data might suffice to identify some potential attackers or other network anomalies. In this application, low latency processing is critical to mitigate attacks. However, some events might go undetected because a subset of the information is missing, and some aggregate results may be incorrect. Furthermore, the state of replicas diverges as they process different inputs.

After a failure heals, previously unavailable data streams are made available again. To ensure that replicas become once again consistent with one another and that client applications eventually receive the complete correct streams, it is important to arrange for each node to correct its internal state and the output it produced during the failure.

2.2 Failure Assumptions

Our approach handles fail-stop failures (*e.g.*, software crashes) of processing nodes, network failures, and network partitions where any subset of nodes lose connectivity to one another. When each node has N replicas (including itself), we tolerate up to $N - 1$ simultaneous node failures. We consider long delays as network failures.

We assume that data sources and clients implement the fault-tolerance protocols described in the next section. This can be achieved by having clients and data sources use a fault-tolerant library or by having them communicate with the system through proxies (or nearby processing nodes) that implement the required functionality. We also assume that data sources, or proxies acting on their behalf, log input tuples persistently (*e.g.*, in a transactional queue [10]) before transmitting them to all replicas that process the corresponding streams. A persistent log ensures that all replicas eventually see the same input tuples, in spite of proxy or data source failures. The fail-stop failure of a data source, however, causes the permanent loss of input tuples that would have otherwise been produced by the data source.

Our scheme is designed for a low level of replication and a low failure frequency. We assume that replicas have spare processing and bandwidth capacity and that they communicate using a reliable, in-order protocol like TCP.

2.3 Design Goals

Our goal is to ensure, for each node, that any data tuple on an input stream is processed within a specified time bound, regardless of whether failures occur on other input streams or not. Among possible ways to achieve this goal, we seek methods that produce the fewest tentative tuples. If $N_{tentative}$ is the number of tentative tuples produced by a node and $Delay_{new}$, the *maximum* delay for that node to process an input tuple and produce a result, our goal is for each node to minimize $N_{tentative}$, subject to $Delay_{new} < X$.

X is a measure of the maximum processing latency that an application or user can tolerate to avoid inconsistency. Different algorithms are possible to convert an end-to-end latency into a per-node delay. We do not discuss this assignment in this paper and assume each node is given X. The constraint on $Delay_{new}$ implies that a node cannot buffer inputs longer than αX, where $\alpha X < X - P$ and P is the normal processing delay. Alternatively, X could express an *added* delay, but we use the former definition in this paper.

Reducing $N_{tentative}$ reduces the amount of resources consumed by downstream nodes in processing tentative tuples. $N_{tentative}$ may also 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.

Our approach ensures that as long as some path of non-blocking operators is available between one or more data sources and a client application, the client receives results. Furthermore, our approach favors stable results over tentative results when both are available. Once failures heal, we ensure that clients receive stable versions of all results, and that all replicas converge to a consistent state. We handle single failures and multiple overlapping (in time) failures.

3. APPROACH

This section describes our replication scheme and underlying algorithms. Each node implements the state machine shown in Figure 2 that has three states: STABLE, UPSTREAM_FAILURE (UP_FAILURE), and STABILIZATION.

As long as all upstream neighbors of a node are producing stable tuples, the node is in the STABLE state. In this state, it processes tuples as they arrive and passes stable results to downstream neighbors. 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.

If one input stream becomes unavailable or starts carrying tentative tuples, a node goes into the UP_FAILURE state, where it tries to find another stable source for the input stream. If no such source is available, the node has three choices to process the remaining available input tuples:

1. *Suspend* processing until the failure heals and the failed upstream neighbors start producing stable data again.
2. *Delay* new tuples for a short period of time before processing.
3. *Process* each new tuple without any delay.

The first option favors consistency. It does not produce any tentative tuples and may be used only for short failures given our goal to process new tuples with bounded delay.

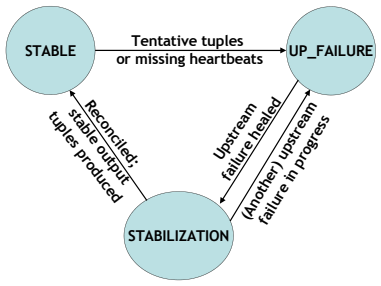

Figure 2: The Borealis state machine.

The latter two options both produce result tuples that are marked "tentative;" the difference between the options is in the latency of results and the number of tentative tuples produced. Section 3.3 discusses the UP_FAILURE state.

A failure *heals* when a previously unavailable upstream neighbor starts producing stable tuples again or when a node finds another replica of the upstream neighbor that can provide the stable version of the stream. Once a node receives the stable versions of all previously missing or tentative input tuples, it transitions into the STABILIZATION state. In this state, if the node processed any tentative tuples during UP_FAILURE it must now reconcile its state and stabilize its outputs. We explore two approaches for state reconciliation: a checkpoint/redo scheme and an undo/redo scheme. While reconciling, new input tuples are likely to continue to arrive. The node has the same three options mentioned above for processing these tuples: suspend, delay, or process without delay. Our approach enables a node to reconcile its state and correct its outputs, while ensuring that new tuples continue to be processed. We discuss the STABILIZATION state in Section 3.4.

Once stabilization completes, the node transitions to the STABLE state if there are no other current failures, or back to the UP_FAILURE state otherwise.

3.1 Data Model

With our approach, nodes and applications must distinguish between stable and tentative results. Stable tuples produced after stabilization may override previous tentative ones, requiring a node to correctly process these amendments. Traditionally, a stream is an append-only sequence of tuples of the form: $(t, a_1, \ldots, a_m)$, where t is a timestamp value and $a_1, \ldots, a_m$ are attribute values [1]. To accommodate our new tuple semantics, we adopt and extend the Borealis data model [3]. In Borealis, tuples take the form:

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

1. **tuple_type** indicates the type of the tuple.
2. **tuple_id** uniquely identifies the tuple in the stream.
3. **tuple_time** is the tuple timestamp. We discuss these timestamps further in Section 3.2.

Traditionally, all tuples are immutable stable insertions. We introduce two new types of tuples: TENTATIVE and UNDO. A tentative tuple is one that *results from processing a subset of inputs and may subsequently be amended with a stable version.* An undo tuple indicates that a suffix of tuples on a stream should be deleted and the associated state of any operators rolled back. As illustrated in Figure 3, the undo tuple indicates the suffix with the tuple_id of the last tuple not to be undone. Stable tuples that follow an undo replace the undone tentative tuples. Applications that do not tolerate inconsistency may thus simply drop tentative and

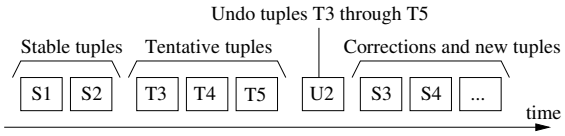

Figure 3: Example of using tentative and undo tuples. U2 indicates that all tuples following tuple with tuple_id 2 (S2 in this case) should be undone.

Tuple type	Description
Data streams	
STABLE	Regular tuple
TENTATIVE	Tuple that results from processing a subset of inputs and may be corrected later
UNDO	Suffix of tuples should be rolled back
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	Signals from SUnion
UP_FAILURE	Entering inconsistent state
REC_REQUEST	Input was corrected, can reconcile state

Table 1: Types of tuples

undo tuples. We use a few additional tuples types in our approach but they do not fundamentally change the data model. Table 1 summarizes the new tuple types.

3.2 Stable State

An operator is *deterministic* if its results do not depend on the times at which its inputs arrive (*e.g.*, the operator does not use timeouts); of course, the results will usually depend on the input data order. If all operators are deterministic, we only need to ensure that replicas of the same operator process data in the same order to maintain consistency; otherwise, the replicas will diverge even without failures.

Since nodes communicate with TCP, tuples never get reordered within a stream and the problem affects only operators with more than one input stream (*e.g.*, Union and Join). We thus need a way to order tuples deterministically across multiple input streams that feed the same operator. The challenge is that tuples on streams may not be sorted on any attribute and they may arrive at significantly-different rates. To compute an order without the overhead of inter-replica communication, we propose a simple data-serializing operator, *SUnion*. SUnion takes multiple streams as input and applies a deterministic sort function on *buckets* of tuples.

SUnion uses tuple_time values to place tuples in buckets of statically defined sizes. The sort function later typically orders tuples by increasing tuple_time values, but other functions are possible. 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 that no tuples with tuple_time smaller than the boundary's tuple_time will be sent after the boundary[1]. Boundary tuples are similar to punctuation tuples [41] or heartbeats [36].

Figure 4 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 (in bucket i+1). These boundary tuples make the bucket *stable* as they guarantee that no tuples are missing from the bucket. Neither of the other buckets can be processed, since both buckets are missing boundary tuples and bucket $i+2$ contains tentative tuples.

[1]If a data source cannot set these values, the first processing node to see the data can act as a proxy for the data source, setting tuple headers and producing boundary tuples.

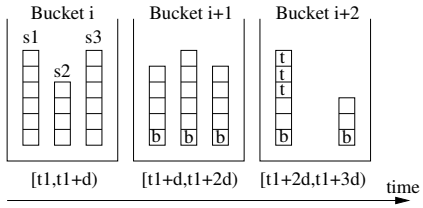

Figure 4: Example of serialization of streams s_1, s_2, and s_3 with boundary interval d. The t's denote tentative inserts and b's denote boundary tuples.

SUnion operators may appear at any location in a query network. Operators must thus set tuple_time values on their output tuples deterministically as these values will affect tuple order at downstream SUnions. Operators must also produce periodic boundary tuples and tuple_time values in boundary tuples must be monotonically increasing. If output tuples are not ordered on tuple_time values, boundary tuples must propagate through the query network to enable downstream operators to produce correct boundary tuples.

SUnion is similar to the Input Manager in STREAM [36], which sorts tuples by increasing timestamp order and deduces heartbeats if applications do not provide them. SUnion, in contrast, ensures that replicas process tuples in the same order, distinguishes failures from delays, offers a flexible availability/consistency trade-off (as we discuss in the next section), and corrects input streams after failures heal. The Input Manager does not make such distinctions. It assumes that delays are bounded.

A natural choice for tuple_time is to use wall clock time. By synchronizing clocks at the data sources, tuples will get processed approximately in the order they are produced. The NTP (Network Time Protocol) [40] is standard today and implemented on most computers and essentially all servers. NTP synchronizes clocks to within 10 ms. Wall-clock time is not the only possible choice, though. In Borealis, any integer attribute can serve to define the windows that delimit operator computations. When this is the case, operators also assume that input tuples are sorted on that attribute and tolerate only limited re-ordering [1]. Hence, using the same attribute for tuple_time as for windows helps enforce the ordering requirement.

SUnion operators delay tuples because they buffer and sort them. This delay depends on three properties of boundary tuples. First, the interval between boundary tuples with increasing tuple_time values as well as the bucket size determine the average buffering delay. Second, the buffering delay further increases with disorder. The increase is bounded above by the maximum delay between a tuple with a tuple_time, t, and a boundary tuple with a tuple_time $> t$. Third, a bucket is stable only when boundary tuples with sufficiently high tuple_time values appear on all streams input to the same SUnion. The maximum differences in tuple_time values across these streams bounds the added delay. Because the query network typically assumes tuples are ordered on the attribute selected for tuple_time, we can expect serialization delays to be small in practice. In particular, these delays should be significantly smaller than the maximum processing delay, X.

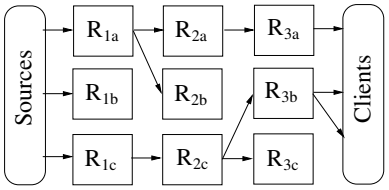

Figure 5: Example of replicated SPEs. R_{ij} **is the** j**'th replica of processing node** i**.**

3.3 Upstream Failure

Each node monitors the availability and consistency of its input streams by periodically requesting heartbeat responses from *each replica* of each upstream neighbor. These responses not only indicate if a replica is reachable but include the states (STABLE, UP_FAILURE, or STABILIZATION) of its output streams. Even though a node is in UP_FAILURE, a subset of its outputs may be unaffected by the failure and may remain in the STABLE state. Additionally, a node monitors the data it receives, namely the identifiers of the last stable and tentative input tuples on each input stream.

With the above information, if an upstream neighbor is no longer in the STABLE state or is unreachable, the node can switch to another STABLE replica of that neighbor and continue receiving data from the correct point in the stream. If no STABLE replica is reachable, the node will try to continue from a replica in the UP_FAILURE state to ensure the required availability. The result of these switches is that any replica can forward data streams to any downstream replica or client and the outputs of some replicas may not be used, as illustrated in Figure 5. We further discuss switching between upstream neighbors in various consistency states in Section 3.5.

To enable such switches, every node buffers its 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; we further discuss buffer management in Section 5.4.

If a node fails to find a STABLE replica to replace an upstream neighbor it can either block or continue processing the available tentative tuples or even continue with a missing input stream. Blocking avoids inconsistency and is thus the best approach for failures shorter than αX. For longer failures, the node must eventually stop blocking new tuples to ensure the required availability. When this occurs, SUnions serialize the available tuples, labelling them as tentative, and buffering them in preparation for future reconciliation (SUnions monitor all input streams). In the example from Figure 4 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 tentative tuples αX time units after the first tuple entered that bucket, SUnion will store and forward the remaining tuples as tentative.

As a node processes tentative tuples, its state may start to diverge. The node can do one of two things: delay new tuples as much as possible or process them without delay. Continuously delaying new tuples reduces the number of tentative tuples produced during failure but it constrains what the node can do during stabilization, as we discuss next.

3.4 Stabilization

A node determines that a failure healed when it is able to communicate with a stable upstream neighbor and receives corrections to previously-tentative tuples (or a replay of previously missing inputs). To ensure eventual consistency, the node must then reconcile its state and stabilize its outputs. This means that the node replaces previously tentative result tuples with stable ones, thus allowing downstream neighbors to reconcile their states in turn. To avoid correcting tentative tuples with other tentative ones, a node reconciles its state only after correcting *all* its input streams. We present state reconciliation and output stabilization techniques in this section. We also present a technique that enables each node to maintain availability (meet the $\text{Delay}_{\text{new}} < X$ requirement) while reconciling its state.

3.4.1 State Reconciliation

Because no replica may have the correct state after a failure and because the state of a node depends on the exact sequence of tuples it processed, we propose that a node reconcile 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 tentative tuples. Both approaches require that the node suspends processing new input tuples while reconciling its state.

Checkpoint/redo reconciliation. In this approach, a node periodically checkpoints the state of its query network when it is in STABLE state. SUnions on input streams buffer input tuples between checkpoints and they continue to do so during UP_FAILURE. These input tuples must be buffered 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, a node suspends all processing and iterates through operators and intermediate queues to make a copy of their states. Checkpoints could be optimized to copy only differences in states since the last checkpoint. We do not investigate this optimization and show, in Section 5.3, that it is actually not needed. To reconcile its state, a node re-initializes operator and queue states from the checkpoint and reprocesses all buffered input tuples. To enable this approach, operators must thus be modified to include a method to take a snapshot of their state or reinitialize their state from a snapshot.

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 tentative tuples and redoing that of their stable counterparts. With undo/redo, SUnions on input streams only need to buffer tentative 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 de-

termine 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 pointers to some location in the input buffer. To produce the appropriate undo tuple, operators must store the last tuple they produced with each set of stream markers.

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, rather than compute stream markers for every tuple, operators may set stream markers periodically. This will increase reconciliation time, however, as re-processing will restart from an inexact marker.

3.4.2 Stabilizing Output Streams

Independently of the approach chosen to reconcile the state, a node stabilizes each output stream by deleting a suffix of the stream (normally all tentative tuples) with a single undo tuple and forwarding corrections in the form of stable tuples. When it receives an undo tuple, an SUnion at a downstream node stabilizes the corresponding input stream by replacing, in its buffer, undone tuples with their stable counterparts. Once all input streams are corrected, SUnions trigger a state reconciliation.

With undo/redo, operators process and produce undo tuples, which simply propagate to downstream nodes. To generate an undo tuple with checkpoint/redo, we introduce a new operator, *SOutput*, that we place on each output stream that crosses node boundary. 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.

Stabilization completes when one of two situations occurs. The node re-processes all previously tentative input tuples *and* catches up with normal execution (i.e., it clears its queues) or another failure occurs and the node goes back into UP_FAILURE. Once stabilization completes, a node transmits a REC_DONE tuple to its downstream neighbors. 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 tentative tuples. We propose to achieve this by using two replicas of a query network: one replica remains in UP_FAILURE state and continues processing new input tuples while the other replica performs the reconciliation. A node could run both versions locally but because we *already use replication*, we propose that replicas use each other as the two versions, when possible. By doing so, we *never* create new replicas in the system. Hence, to ensure availability, before reconciling its state, a node must find another replica and request that it postpone its own reconciliation.

It is up to each downstream node to detect when any one of its upstream neighbors goes into the STABILIZATION state and stops producing recent tuples in order to produce corrections. The downstream node then remains connected to that replica to correct its input stream while at the same time, connecting to another replica that is still in UP_FAILURE state (if possible). The downstream node processes *both* streams in parallel, until it receives a REC_DONE tuple on the corrected stream. At this point, it enters the STABILIZATION state, in turn. SUnion considers that tentative tuples between an UNDO and a REC_DONE correspond to the old failure while tentative tuples that appear after the REC_DONE correspond to a new failure. We discuss how a node produces the correct REC_DONE tuple in spite of failures during its 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 tentative tuples but may eventually break the availability requirement. Processing new tuples during reconciliation increases the number of tentative tuples but a node may still attempt to reduce their number by delaying new tuples as long as possible. We compare these alternatives in Section 5.1.

3.4.4 Failed Node Recovery

A failed node restarts from an empty state and refuses new clients until it processes sufficiently many tuples to reach a consistent state. This approach is possible when operators are *convergent capable* [25]: *i.e.*, they keep a finite state that is also updated in a manner that always converges back to a consistent state. Our schemes could be extended to other types of operators by recovering using a combination of persistent checkpoints and logging.

3.5 Analysis

We now 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 becomes s after traversing some sequence of operators, called a *path*. The union of paths that connect a set of sources to a destination (a client or an operator), forms a *tree*. A tree is valid if paths that traverse the same operator also traverse the same replica of that operator. A valid tree is *stable* if it contains all data sources that contribute to the stream received by the destination. A stable tree 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 *tentative*. Otherwise, the tree is *blocking*. Figure 6 illustrates each type of tree.

PROPERTY 1. *In a static failure state, if there exists a stable tree, a destination receives stable tuples. If only tentative trees exist, the destination receives tentative tuples from one of the tentative trees. In both cases, the destination receives results within at most a kX time-unit delay, where X is the delay assigned to each SUnion operator and k is the number of SUnions on the longest path in the tree. In other cases, the destination may block.*

The above property comes from the ability of downstream nodes to monitor and switch upstream neighbors, preferring stable ones over those in UP_FAILURE state and those in UP_FAILURE state over no input at all. We study the delay properties in Section 5, where we assume that the number

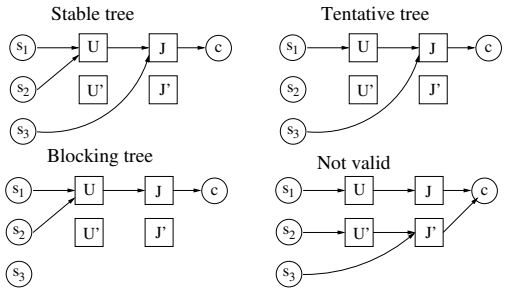

Figure 6: Example trees for a query network with three sources, one client, a Union, and a Join. $\{s_1, s_2, s_3\}$ contributes to the stream received by c. Each operator has two replicas.

of SUnions is equal to the number of nodes. If this is not the case, the delay assigned to a node must be divided among the sequence of SUnions at the node.

PROPERTY 2. *Switching between trees never causes duplicate results and may only lose tentative tuples.*

We discuss this property by examining each possible neighbor-switching scenario:

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 a neighbor in* UP_FAILURE *state to a stable upstream neighbor*: In this situation, the downstream node indicates the identifiers of the last stable and tentative tuples it received. This allows the new upstream neighbor to stabilize the stream and continue with stable tuples.

3) *Switching to an upstream neighbor in* UP_FAILURE *state*: Because nodes cannot undo stable tuples, the new upstream and downstream pair may have to continue processing tuples while in mutually inconsistent states, which can lead to duplicate or missing results. We choose to avoid duplications as this leads to fewer tentative tuples. We add a second timestamp, t_max to tuples. t_max of a tuple p is the tuple_time of the *most recent input tuple* that affected p. The new upstream node forwards only output tuples that have a t_max greater than the highest t_max that the downstream node previously received. These tuples necessarily result from processing at least a partially non-overlapping sequence of input tuples. Other techniques are possible.

4) If an upstream neighbor is in the STABILIZATION state, a node treats the incoming stream as redundant information that serves to correct input streams in the background.

PROPERTY 3. *As long as one replica of each processing node never fails, assuming all tuples produced during a failure are buffered, when all failures heal, the destination receives the complete stable stream.*

After a failure heals, each node reconciles its state and stabilizes its output, letting its downstream neighbors correct their inputs and reconcile in turn. This process propagates all the way to the clients.

PROPERTY 4. *Stable tuples are never undone.*

We show that our approach handles failures during failures and recovery without the risk of undoing stable tuples.

Undo/redo reconciliation: As soon as an operator receives a tentative tuple, it starts labeling its output tuples as tentative. Therefore, undoing tentative tuples can never cause a stable output to be undone. When reconciling, SUnions produce undo tuples followed by the stable versions of tuples processed during the failure. Any new tentative input tuples will thus be processed after the undo and stable tuples such that any new failure will follow the reconciliation, without affecting it. While an undo tuple propagates on a stream, if a different input stream becomes tentative, and both streams merge at an operator, the operator could see the new tentative tuples before the undo tuple. In this case, 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* going back to processing the new tentative tuples. The operator thus produces an undo tuple followed by stable tuples that correct the first failure, followed by the tentative tuples from the new failure. Once again, the new failure appears to occur after stabilization.

Checkpoint/redo: SOutput guarantees that stable tuples are never undone. When restarting from a checkpoint, SOutput enters a "duplicate elimination" mode. It remains in that state and continues waiting for the same last duplicate tuple until it produces the undo tuple, even if another checkpoint or recovery occurs. After producing the undo, SOutput goes back to its normal state, where it remembers the last stable tuple that it sees and saves the identifier of that tuple during checkpoints.

In both cases, if a new failure occurs before the node had time to catch up and produce a REC_DONE tuple, SOutput forces a REC_DONE tuple between the last stable and first tentative tuples that it sees.

4. IMPLEMENTATION

To implement our scheme in Borealis, in addition to inserting *SUnion* and *SOutput* operators into query networks, we add a *Consistency Manager* and an *HA* ("high availability") component to each SPE node. Figures 7 and 8 illustrate these modifications (arrows indicate communication between components).

HA monitors all the replicas of a node and those of its upstream neighbors. It informs the query processor of changes in the states of their outputs. To modify the data path, nodes send each other subscribe and unsubscribe messages.

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 asks a partner to suspend its own reconciliation and chooses whether to use undo/redo or checkpoint/redo. For undo/redo, the Consistency Manager injects UNDO_START tuples on input streams of affected SUnion operators. 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 UP_FAILURE, by producing an UP_FAILURE 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 and output streams.

We also require operators to implement a simple API. For

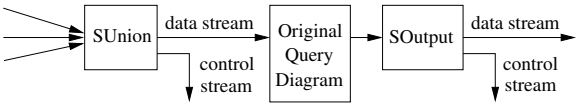

Figure 7: Modified query network.

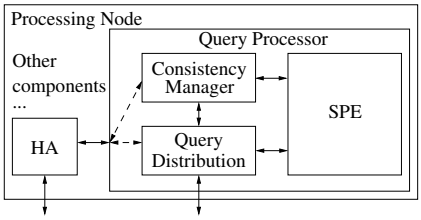

Figure 8: Extended software node architecture.

checkpoint/redo, operators need the ability to take snapshots and recover their state ((un)packState methods). For undo/redo, operators must be able to correctly process undo tuples. At runtime, they must compute stream markers and remember the last tuple they output. 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 the method findOldestTimestamp() that returns the oldest timestamp that the operator can still produce. This value is typically 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.

5. EVALUATION

In this section, we evaluate the performance of our fault-tolerance protocol through experiments with our prototype implementation. All single-node experiments were performed on a 3 GHz Pentium IV with 2 GB of memory running Linux (Fedora Core 2). Multi-node experiments were performed by running each pair of node replicas on a different machine. All machines were 1.8 GHz Pentium IV's or faster with greater than 1 GB of memory.

Our basic experimental setup is the following. We run a query network composed of three input streams, an SUnion that merges these streams into one, a Join that serves as a generic query network with a 100 tuple state size, and an SOutput. The aggregate input rate is 3000 tuples/s. We create a failure by temporarily disconnecting one of the input streams without stopping the data source. After the failure, we send all missing tuples while continuing to stream new tuples. X is 3 s. α is 0.9 (so αX is 2.7 s). Each result is an average of at least three experiments.

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 $\text{Delay}_{\text{new}}$ and $\text{N}_{\text{tentative}}$ different strategies regarding suspending, delaying, and processing new tuples during UP_FAILURE and STABILIZATION. As we point out, some combinations are unviable as they break the availability requirement for sufficiently long failures. 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.

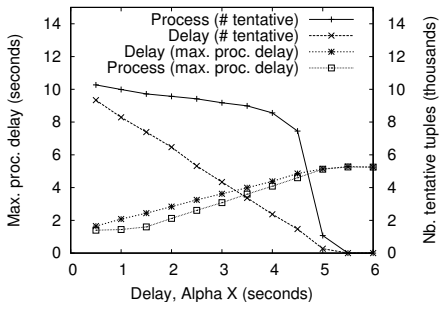

Figure 9: Delaying tuples during UP_FAILURE reduces $\text{N}_{\text{tentative}}$. Y-Axes: left $\text{Delay}_{\text{new}}$, right $\text{N}_{\text{tentative}}$. Failure duration: 5 s.

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 is therefore always our first line of defense. When a failure exceeds αX, however, a node must restart processing new tuples to satisfy the availability requirement. It can either continuously delay new tuples by αX or catch-up and process new tuples almost as they arrive. We call these alternatives Delay and Process and examine their impact on $\text{Delay}_{\text{new}}$ and $\text{N}_{\text{tentative}}$.

We cause a 5 s failure, vary αX from 500 ms to 6 s, and observe $\text{Delay}_{\text{new}}$ and $\text{N}_{\text{tentative}}$ until after STABILIZATION completes. Figure 9 shows the results. From the perspective of our optimization, Delay appears better than Process as it leads to fewer tentative tuples. Indeed, 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 tentative tuples almost proportional to the failure duration. The $\text{N}_{\text{tentative}}$ graph approximates a step function. In contrast, Delay reduces the number of tentative tuples proportionally to αX. With both approaches, $\text{Delay}_{\text{new}}$ increases linearly with αX.

For sufficiently long failures, however, reconciliation itself may last longer than X. To avoid breaking the availability requirement, a node must thus continue processing new tuples while reconciling. It can do so in one of several ways. During the failure, the node can either delay new tuples (Delay) or process them without delay (Process). During STABILIZATION the node can either suspend new tuples (Suspend), or have a second version of the SPE continue processing them with or without delay (Delay or Process). Our goal is to examine all six combinations and determine the failure durations when each one produces the fewest tentative tuples without breaking the availability requirement.

Figure 10 shows $\text{Delay}_{\text{new}}$ and $\text{N}_{\text{tentative}}$ for each combination and for increasing failure durations. We only show results for failures up to 1 minute. Longer experiments continue the same trends. In this experiment, we increase the input rate to 4500 tuples/s to emphasize differences between approaches.

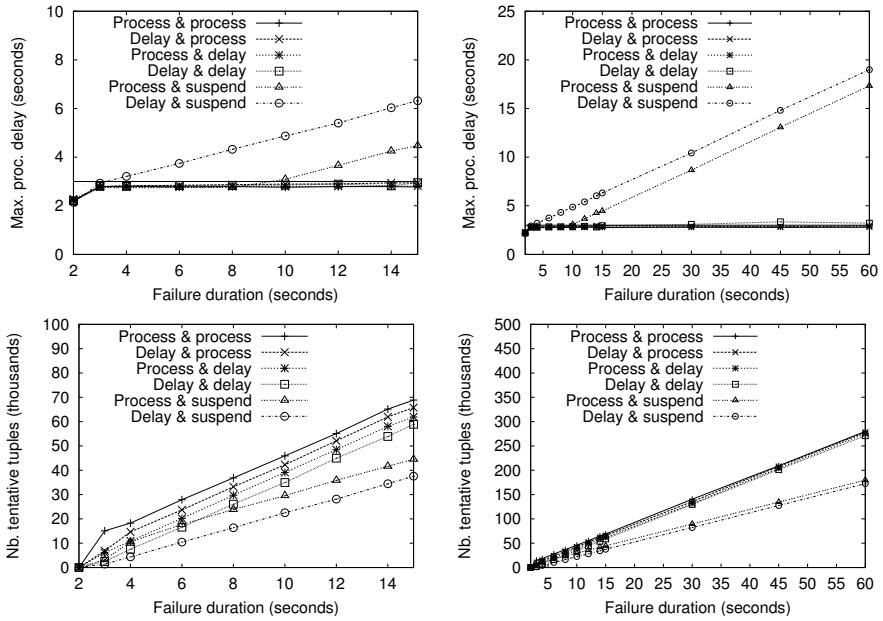

Figure 10: $\text{Delay}_{\text{new}}$ **(top) and** $\text{N}_{\text{tentative}}$ **(bottom) for each combination of delaying, processing, and suspending during** UP_FAILURE **and** STABILIZATION**. Each approach offers a different consistency-availability trade-off. X-axis starts at 2 s. Graphs on the right show results for longer failures.**

Because blocking is optimal for short failures, all approaches block for $\alpha X = 2.7$ s and produce no tentative tuples for failures below this threshold. Delaying tuples in UP_FAILURE and suspending them during STABILIZATION (Delay & Suspend) is unviable for failures longer than 3 s because it breaks the $\text{Delay}_{\text{new}} < X$ requirement as reconciliation last longer than 300 ms. (Figure 10(top)). Therefore, this combination is of no interest because it never wins and cannot be used for long failures.

Continuously processing new tuples during both UP_FAILURE and STABILIZATION (Process & Process) ensures that the maximum delay always remains below αX independently of failure duration. This combination, however, produces the most tentative tuples as it produces them for the duration of the whole failure and reconciliation. We can reduce the number of tentative tuples without hurting $\text{Delay}_{\text{new}}$, by delaying new tuples during STABILIZATION (Process & Delay), during UP_FAILURE, or in both states (Delay & Delay).

For short failures, however, Process & Suspend may win over Delay & Delay. If reconciliation is longer than αX (for $D > 6$ s in the experiment), Process & Suspend produces fewer tentative 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, which is a small, barely significant window.

Hence to meet the availability requirement for longer failures, nodes must process new tuples not only during UP_FAILURE but also during STABILIZATION. Nodes can produce fewer tentative tuples, however, by always running on the verge of breaking that requirement.

5.2 Multiple Nodes

We now examine which of the above combinations meets the required availability while producing the fewest tentative tuples in a distributed SPE. We cause a 15 second failure at the input of a chain of 1 to 4 SPEs. Once the failure heals, the nodes reconcile their states in sequence: a node produces boundary tuples only after it goes back into STABLE state, while its downstream neighbors can start reconciling only after receiving these boundary tuples. We reduce the state of the joins to 50 tuples to speed-up the experiments.

Figure 11(top) shows the maximum end-to-end processing delay for new tuples. The Process & Process combination has the lowest $\text{Delay}_{\text{new}}$. The delay is equal to only αX plus the normal processing delay through the chain. Delay & Delay leads to a slightly worse availability as $\text{Delay}_{\text{new}}$ increases by αX for each node in the sequence. Both combinations, however, keep the end-to-end delay within the required kX, where k is the number of nodes in the chain. Process & Suspend once again is clearly unviable. $\text{Delay}_{\text{new}}$ is the sum of the stabilization delays of all nodes in the chain. This delay increases for each consecutive node as it undoes and redoes more tuples than its upstream neighbor.

Figure 11(bottom) shows $\text{N}_{\text{tentative}}$ received by the client application. With Process & Process, $\text{N}_{\text{tentative}}$ increases with the length of the chain because all nodes produce tentative tuples during STABILIZATION, which occurs in sequence at each node. Interestingly, Delay & Delay not only does not provide any benefit but can even hurt when compared with no delay. Indeed, 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 significantly more tuples than the savings during UP_FAILURE.

Overall, for a chain of nodes, Process & Process is clearly the best approach as it maximize availability and produces the fewest tentative tuples.

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

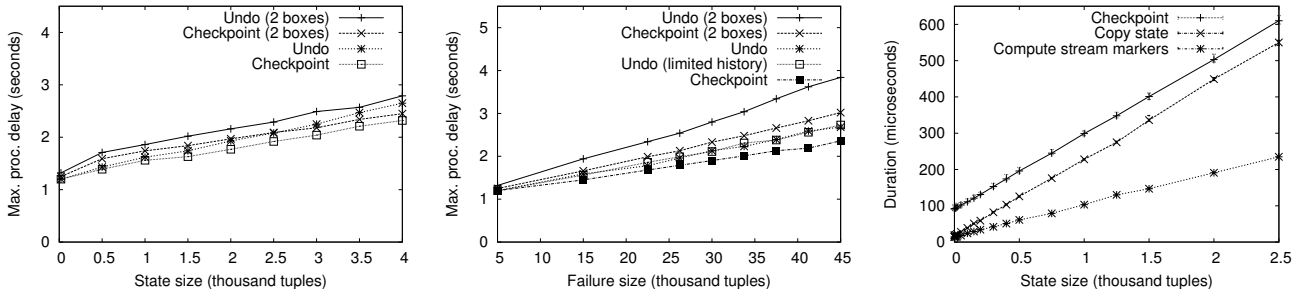

Figure 12: Performance and overhead of checkpoint/redo and undo/redo reconciliations. $\text{Delay}_{\text{new}}$ for increasing state size (left). $\text{Delay}_{\text{new}}$ for increasing failure size starting at 5000 tuples (middle). CPU Overhead (right). Checkpoint/redo is faster than undo/redo but checkpoints can be expensive.

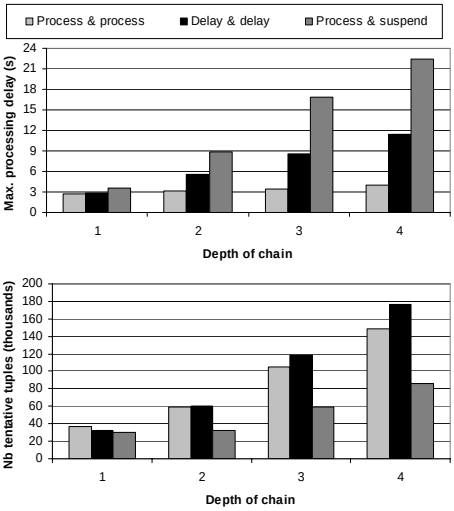

Figure 11: Effects of path length on $\text{Delay}_{\text{new}}$ (top) and $\text{N}_{\text{tentative}}$ (bottom). Process & Suspend is unviable. Process & Process achieves the best availability without increased inconsistency.

5.3 Reconciliation

We now compare the overhead and performance of checkpoint/redo and undo/redo reconciliation. Overheads due to SUnion operators are examined in the next section. Table 2 summarizes the analytical 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. p_{proc} is the time an operator takes to process a tuple. We assume p_{proc} is constant but it may increase with operators' state sizes.

$\text{Delay}_{\text{new}}$ is the normal processing delay, P, plus the reconciliation time. For checkpoint/redo, the reconciliation time is the sum of Sp_{copy}, the time to copy the state with size S, and $(D + 0.5l)\lambda p_{\text{proc}}$, the average time to reprocess all tuples since the last checkpoint before failure. D is the failure duration, l is the interval between checkpoints, and λ is the *aggregate tuple rate on all input and intermediate streams.* For undo/redo, reconciliation consists of processing the undo history up to the correct stream markers and reprocessing all tuples since then. Producing an undo message takes a negligible time. We assume that the number of tuples necessary to rebuild an operator state is equal to the state size and that stream markers are computed ever l time units. The number of tuples in the undo log that must be processed backward then forward is thus: $(D + 0.5l)\lambda + S$. Hence, we expect checkpoint/redo to perform better but the difference should appear only for a large query network state size.

Figure 12 shows the experimental $\text{Delay}_{\text{new}}$ as we increase the state size, S, of the query network (left) or the number of tuples to re-process *i.e.*, $D\lambda$ (middle). 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, we keep the tuple rate low at 1000 tuples/s. When we vary the tuple rate, we keep the state size at only 20 tuples.

Undo/redo takes longer to reconcile primarily because it must rebuild the state of the query network (Sp_{proc}) rather than recopy it (Sp_{copy}), as shown in Figure 12(left). Interestingly, even when we keep the state size small and vary the number of tuples to reprocess (Figure 12(middle)), checkpoint/redo 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 tentative 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 UP_FAILURE. For checkpoint/redo, because we checkpoint every 200 ms, we 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.

As shown in Figure 12(left and middle), for both approaches, splitting the state across two operators in series (curves labeled "2 boxes"), simply doubles λ and increases curve slopes.

In theory, checkpoint/redo has higher CPU overhead than undo/redo because checkpoints are more expensive than scanning the state of an operator to compute stream markers (Figure 12(right)). However, because a node has time to checkpoint its state when going into UP_FAILURE state, it can *perform checkpoints only at that point* and avoid the

Boundary 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.

overhead of periodic checkpoints at runtime. Stream markers can also be computed only once a failure occurs. Hence both schemes can avoid overhead in the absence of failures.

Given that we checkpoint the state when entering UP_FAILURE, $l = 0$. Hence, the memory overhead for checkpoint/redo is only $S + D\lambda_{in}$, the state size plus the input tuples that accumulate during the failure (λ_{in} is the aggregate *input* rate). Even if we assume that we need no more than S tuples to rebuild the state, the memory overhead for undo/redo is higher because we need to buffer tuples on all streams that feed stateful operators. $\lambda_{\text{stateful}}$ will most frequently be significantly greater than λ_{in}.

Checkpoint/redo thus appears superior to undo/redo both in terms of reconciliation time and memory overhead. 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 Overhead and Scalability

In addition to undo and checkpoint overheads, SUnions are the main cause of overhead. If the sorting function requires the operator to wait until a bucket is stable before processing tuples, the processing delay increases linearly with the boundary tuple interval (we assume this interval is equal to the bucket size). Table 3 shows the average end-to-end delay from nine 20 s experiments and increasing bucket sizes. The memory overhead increases proportionally to the number of SUnion operators, bucket sizes, and the rate of tuples that arrive into each SUnion.

Other overheads imposed by our scheme are negligible. Operators must check tuple types and must process boundary tuples. The former is negligible while the latter is equivalent to the overhead of computing stream markers. SOutput must also save the last stable tuple that it sees in every burst of tuples that it processes.

Our approach relies on replication. It increases resource utilization proportionally to the number of replicas. These replicas, however, can actually improve runtime performance by forming a content distribution network, where clients and nodes connect to nearby upstream neighbors rather than a single, possibly remote, location.

In this paper, we assume that tuples produced during failure and recovery are logged in output buffers and inside SUnions on input streams. Under normal operation, a node can truncate its output buffers once all replicas of all downstream neighbors acknowledge either receiving or fully processing a prefix of tuples. Both techniques are acceptable. As discussed in [25], acknowledging only processed tuples has the advantage that input tuples necessary to rebuild the latest consistent state are stored at upstream neighbors, which speeds-up recovery of failed nodes. A similar approach can be used to truncate buffers during failures, preserving only enough tuples to rebuild the latest consistent state and correct the most recent tentative tuples. To truncate buffers, a node must hear at least from one downstream replica during a failure. Otherwise, a node may have to use conservative estimates to truncate its buffers.

In this paper, we assume that operators are convergent-capable but our techniques can be extended to support arbitrary operators. For such operators, however, when sufficiently long failures occur, the system must either drop tuples at system input, or replicas must communicate with each other to reach a mutually consistent state after failures heal. We plan to explore such extensions in future work.

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 techniques either do not address network failures [25] or strictly favor consistency by requiring at least one fully connected copy of the query network to exist to continue processing [35]. Some techniques use punctuation [41], heartbeats [36], or statically defined slack [1] to tolerate bounded disorder and delays. These approaches, however, block or drop tuples when disorder or delay exceed expected bounds. Another approach, developed for publish-subscribe systems tolerates failures by restricting all processing to "incremental monotonic transforms" [37].

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 availability in the presence of network partitions [11]. Eager replication favors consistency by having a majority of replicas perform every update as part of a single transaction [20, 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, 42], such as preserving only the most recent version of a record [22]. It is unclear how one could define such rules for an SPE and reach a consistent state. Other replication approaches use tentative transactions during partitions and reprocess transactions possibly in a different order during reconciliation [22, 39]. With these approaches, all replicas eventually have the same state and that state corresponds to a single-node serializable execution. Our approach applies the ideas of tentative 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.*, resource utilization) [32, 33]. In contrast, we explore consistency/availability trade-offs in the face of failures and ensure eventual consistency.

Workflow management systems (WFMS) [5, 4, 24] share similarities with stream processing engines. Existing WFMSs, however, typically commit the results of each execution step (or messages these steps exchange) in a central highly-available storage server [26] or in persistent queues [4]. Some approaches allow replication of the central data server using standard lazy replication [4]. They support disconnection by locking activities prior to disconnection [5].

Approaches that reconcile state after a failure using combinations of checkpoints, undo, and redo are well known [18, 22, 23, 29, 39]. 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

We presented a replication-based approach to fault-tolerant stream processing that handles node failures, network failures, and network partitions. Our approach uses a new data model that distinguishes between stable tuples and tentative tuples, which result from processing partial inputs and may later be corrected. Our approach favors availability but guarantees eventual consistency. Additionally, while ensuring that each node processes new tuples within a predefined delay, X, our approach reduces the number of tentative tuples, when possible. To ensure consistency at runtime, we introduce a data-serializing operator called SUnion. To regain consistency after failures heal, nodes reconcile their states using either checkpoint/redo or undo/redo.

We implemented the approach in Borealis and showed several experimental results. For short failures, SPE nodes can avoid inconsistency by blocking and looking for a stable upstream neighbor. For long failures, nodes need to process new inputs both during failure and stabilization to ensure the required availability. Checkpoint/redo leads to a faster reconciliation at a lower cost compared with undo/redo.

Many stream processing applications prefer approximate results to long delays but eventually need to see the correct output streams. It is important that failure-handling schemes meet this requirement. We view this work as an important first step in this direction.

8. ACKNOWLEDGMENTS

We thank Mehul Shah and Jeong-Hyon Hwang for helpful discussions. This material is based upon work supported by the National Science Foundation under Grant No. 0205445. M. Balazinska is supported by a Microsoft Fellowship.

9. 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] Abadi et al. The design of the Borealis stream processing engine. Technical Report CS–04-08, Department of Computer Science, Brown University, Jan. 2005.
[4] 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.
[5] 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.
[6] 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.
[7] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, May 2000.
[8] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, June 2003.
[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, June 2002.
[10] P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *SIGMOD*, June 1990.
[11] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.
[12] 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.
[13] S. Chandrasekaran and M. J. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *30th VLDB*, Sept. 2004.
[14] Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, Jan. 2003.
[15] Cherniack et al. Scalable distributed stream processing. In *CIDR*, Jan. 2003.
[16] C. Cranor, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: A stream database for network applications. In *SIGMOD*, June 2003.
[17] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, June 2003.
[18] 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.
[19] Feamster et al. Measuring the Effects of Internet Path Faults on Reactive Routing. In *ACM Sigmetrics - Performance 2003*, June 2003.
[20] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841 – 860, Oct. 1985.
[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] R. E. Strom. Fault-tolerance in the SMILE stateful publish-subscribe system. In *DEBS*, May 2004.
[38] N. Tatbul, U.Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *29th VLDB*, Sept. 2003.
[39] Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th SOSP*, Dec. 1995.
[40] The NTP Project. NTP: The Network Time Protocol. `http://www.ntp.org/`.
[41] P. A. Tucker and D. Maier. Dealing with disorder. In *MPDS*, June 2003.
[42] R. Urbano. *Oracle Streams Replication Administrator's Guide, 10g Release 1 (10.1)*. Oracle Corporation, Dec. 2003.