**Jeong-Hyon Hwang, Magdalena Balazinska,
Alexander Rasin, Ugur Cetintemel,
Michael Stonebraker, and Stan Zdonik**

A Comparison of Stream-Oriented High-Availability Algorithms

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-03-17
September 2003

A Comparison of Stream-Oriented High-Availability Algorithms*

Jeong-Hyon Hwang
Brown University
jhhwang@cs.brown.edu

Magdalena Balazinska
MIT
mbalazin@lcs.mit.edu

Alexander Rasin
Brown University
alexr@cs.brown.edu

Uğur Çetintemel
Brown University
ugur@cs.brown.edu

Michael Stonebraker
MIT
stonebraker@lcs.mit.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

Abstract

Recently, significant efforts have focused on developing novel data-processing systems to support a new class of applications that commonly require sophisticated and timely processing of high-volume data streams. Early work in stream processing has primarily focused on stream-oriented languages and resource-constrained, one-pass query-processing. High availability, an increasingly important goal for virtually all data processing systems, is yet to be addressed.

In this paper, we first describe how the standard high-availability approaches used in data management systems can be applied to distributed stream processing. We then propose a novel stream-oriented approach that exploits the unique data-flow nature of streaming systems. Using analysis and a detailed simulation study, we characterize the performance of each approach and demonstrate that the stream-oriented algorithm significantly reduces runtime overhead at the expense of a small increase in recovery time.

1 Introduction

High availability is a key goal of virtually all existing data management systems [8, 17, 19, 21]. As critical applications move on to the Internet, providing highly available services that seamlessly and quickly recover from failures, is becoming increasingly more important. The standard approach to achieving high-availability (HA) involves introducing spare *backup* servers that take over the operation of *primary* servers when the latter fail. In this approach, often called *process pairs* [3, 9], each primary server periodically sends checkpoint messages to its backup. Upon the failure of the primary, the backup immediately takes over from the most recent checkpoint. Process pairs provide single-fault resilience and a short recovery time but impose a high runtime overhead due to checkpoints (added processing delays and bandwidth utilization).

Recently there has been significant activity in the development of novel data processing engines: stream processing engines [1, 2, 4, 5, 15, 18]. The goal of these engines is to better serve a new class of data processing applications, called *stream-based applications*, where data is *pushed* to the system in the form of streams of tuples, and queries are *continuously* executed over these streams. These applications include sensor-based environment monitoring (car traffic, air quality), financial applications (stock-price monitoring, ticker failure detection), asset tracking, and infrastructure monitoring (computer networks, bridges).

Many stream-based applications are inherently distributed. Significant gains in scalability and efficiency can be achieved if multiple distributed servers collectively process and aggregate data streams as they are routed from their points of origin to the target applications. To this end, recent attention has also been geared toward extending stream-processing engines to distributed environments [5, 7].

To achieve high availability in a distributed stream-processing system (DSPS), one needs to devise a set of algorithms that perform the following tasks: (1) periodically and incrementally save (or replicate) the state of processing nodes, (2) detect failures, (3) choose recovery nodes, (4) regenerate the missing state from the saved state when a failure occurs, and (5) handle network partitions. In this paper, we address (1) and (4). We propose a new mechanism *to efficiently save the state of any processing node in a DSPS and re-generate it when a failure occurs.*

We first describe how two process-pair-based algorithms can be adapted to provide high availability in DSPSs: (1) *passive standby*: In this approach, each primary server (a.k.a. node) periodically sends checkpoint messages to a backup node. For our purposes, these messages contain

*This material is based upon work supported by the National Science Foundation under Grant No. 0205445 and IIS-0086057. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

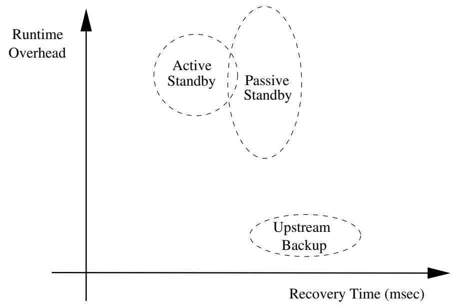


Figure 1. Runtime overhead and recovery time for various high-availability approaches

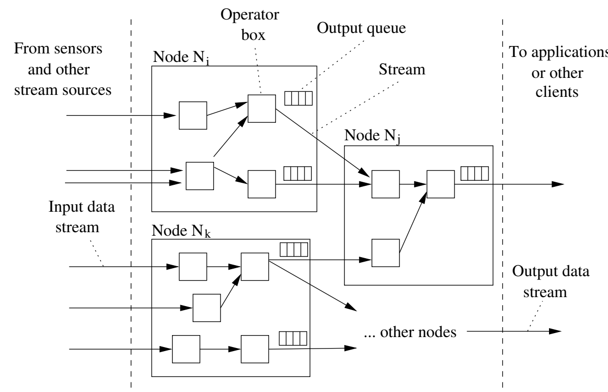


Figure 2. Query execution in a DSPS: Multiple nodes collaborate in solving a query. Each node runs a subset of operators

the state of each operator and the contents of operator input queues. If the primary fails, the backup takes over from the last checkpoint; (2) *active standby*: In this second variant of process-pairs, each primary node also has a backup node. Unlike passive standby, the secondary processes all tuples in parallel with the primary.

Given the high volume of data and the high processing rate found in streaming applications, the overhead of process-pair-based approaches renders them heavyweight for stream processing systems. We therefore propose a new approach, called *upstream backup*, that exploits the distributed and streaming nature of the processing in a DSPS to reduce the runtime operation overhead while trading off only a small fraction of the recovery time performance. In this approach, upstream nodes (in the processing flow) act as backups for their downstream neighbors by preserving tuples in their output queues until their downstream neighbors have processed them. If any of these neighbors fails, upstream nodes replay the logged tuples on a failover node.

Figure 1 illustrates the basic differences among the three approaches. The traditional process-pair approaches (active and passive standby) typically incur high overhead at runtime to achieve short recovery times. The new upstream backup approach, in contrast, incurs relatively low run-time cost at the expense of longer recovery times. Recovery times are in the order of milliseconds for all three approaches.

We use analysis and a detailed simulation study to quantitatively characterize these tradeoffs and to comparatively evaluate the approaches with respect to the overheads they incur during runtime and recovery. We find that upstream backup significantly reduces runtime overhead compared with both types of process-pair-based approaches. We find, on average, a 95% reduction in bandwidth utilization for high availability purposes. Using upstream nodes as backups increases recovery time especially compared with active standby. Interestingly, we find that the recovery time of upstream backup is almost identical to that of passive standby (both approaches re-transmit tuples from upstream nodes through a recovery node and re-process on average half a checkpoint interval). The gains in runtime overhead therefore more than justify a small increase in recovery time.

The rest of this paper is organized as follows. We provide some background on stream-processing in Section 2 and describe the high-availability problem in Section 3. We present the three approaches in Section 4 and evaluate them in Section 5. We summarize related work and conclude in Sections 6 and 7.

2 Background

A data *stream* is a continuous sequence of tuples generated by a data source. Unlike the "process-after-store" data in traditional database systems, streaming tuples are generated in real time and need to be processed as they arrive, before being stored. Several recent projects [1, 2, 5] aim to address this fundamental difference and are building stream-processing systems. We use the Aurora data stream manager [1] as our reference model. Our results, however, are general and applicable to other stream processing models.

Aurora is a data-flow-oriented system where processing is represented by a *boxes*-and-*arrows* paradigm popular in most process-flow and work-flow systems. In an Aurora node, tuples flow through a *query-network*: a loop-free, directed graph of processing operators (also called *boxes*), such as shown in Figure 2. Ultimately, output streams are presented to *applications*. Aurora queries are specified by application developers (within applications) or users by means of a GUI tool. In both cases, queries are built from a standard set of well-defined operators [1].

Aurora is designed to run on a single server. Aurora* and Medusa [7] are two DSPSs that use Aurora as their core single-site processing engine. Aurora* and Medusa take queries destined to Aurora and partition them across multiple computing nodes and even multiple administrative domains. Figure 2 shows an example of a query distribution that would occur in a DSPS such as Aurora* or Medusa.

In this figure, multiple streams arrive from various stream sources. These streams are first processed by a set of operators at nodes N_i and N_k. Streams resulting from this processing are then transmitted to node N_j, and to other nodes. Hence, each node can have multiple output streams and each output stream can be forwarded to multiple downstream neighbors. Since messages flow on a path from N_i to N_j, N_i is said to be *upstream* of N_j, and N_j is said to be *downstream* of N_i (same for N_k and N_j). Eventually, tuples entering N_j are processed by operators at N_j and resulting tuples are forwarded to client applications, possibly on multiple streams.

Streams crossing node boundaries are associated with *output queues*, which are used to store tuples temporarily. Each high-availability approach uses these output queues differently, as discussed further in Section 4. Operator also have input queues through which they receive tuples. These queues are considered to be part of each operator's state and are not represented in the figure.

3 Problem Description and Assumptions

Some streaming applications are more tolerant to failures than most data processing applications. If some data is lost, say in a sensor-based environmental monitoring application, the application can safely continue most of the time. Other streaming applications, however, cannot tolerate data loss, requiring that all tuples get processed correctly. Most financial applications fall into this latter category. Supporting such applications hence requires the development and implementation of algorithms for high availability, where the system can recover from node failures without lost tuples or failed queries. The recovery must also be relatively "fast", as most streaming applications operate on data arriving in real time and expect results to be produced in real time.

In this paper, we investigate high-availability approaches for a DSPS to survive any single failure without losing tuples *and* without stopping the processing of any query. Although we focus on single-fault tolerance, we argue that our approaches are easily extensible to higher degrees of fault tolerance and provide hints toward this direction.

We describe our problem more precisely as follows. Consider a boxes-and-arrows query-network embedded in an overlay network of nodes connected over the Internet. Each node N_i contains a subset of operators from the query-network. In case of a failure of any node N_i, the goal of the high-availability algorithm is to be able to re-start the operators that were running at N_i on a live node N_j and resume the processing in a manner that results in no lost tuples.

In our analysis, we make the following simplifying assumptions. First, we assume there are no network partitions. Second, we assume that there are no cycles in the distribution of a query-network across nodes. In reality, we can avoid cycles by prohibiting their creation when queries are first set-up and by constraining the location where parts of any query can be moved upon failure or load re-balancing. Avoiding cycles prevents a node from accidently becoming its own backup. Third, the focus of this paper is on techniques to save and recover the state of processing nodes. We consider failure detection and choice of recovery nodes as a separate aspect of high availability that is a research issue on its own and is outside the scope of this paper. Fourth, in this initial work, we only address operators that operate on individual tuples; we do not support operators that require windows or persistent storage. Therefore we support *filter*, *map*, and *union* operators from Aurora's language SQuAl [1]. We plan to address the remaining operators in future work. Fifth, we assume that data sources themselves are capable of temporarily storing the tuples they produce until they receive an acknowledgment from the DSPS.

4 High-Availability Approaches

In this section, we describe three alternate approaches to achieve single-fault tolerance in a DSPS. We first describe passive standby, an adaptation of the process-pairs model to stream processing, where the backup server is passive. We then introduce our new algorithm called upstream backup. We finally describe active standby, the second adaptation of the process-pairs models that also relies on some concepts introduced in upstream backup. We discuss how each technique can be extended to support multiple failures. We finally address the issue of duplicate tuple generation during recovery.

4.1 Passive Standby

In this section, we discuss process pairs with passive backup and how they can be applied to a DSPS. We call our adapted approach *passive standby*. The Tandem system [21], which aims at providing nonstop transaction processing, is one example of system that achieves single-fault tolerance through process pairs [3, 9]: a primary process is associated with a backup process that will take over when the primary process fails. To keep the state of both processes synchronized, the primary sends checkpoint messages to the backup at critical points during execution.

The main advantage of process pairs is a short failure recovery time, achieved by having the backup hold a complete and recent snapshot of the primary's state. The main drawback is the bandwidth required to the periodic transmission of possibly large checkpoint messages. Additionally, in passive standby, the primary sends checkpoint messages to the backup *synchronously* to ensure state consistency at the expense of added delays in normal operations. For instance, the primary does not commit until the checkpointed data is available at the backup [19]. It is possible to reduce the processing delays by making the primary and backup run asynchronously: the primary does not wait for acknowledgments from the backup before proceeding past a checkpoint.

This approach, however, may lead to data losses or state inconsistencies if the primary fails while sending a checkpoint message to the backup.

To adapt the passive-standby model to a DSPS, we associate a standby node with each primary node and handle failures at the granularity of nodes. Each checkpoint message should thus contain the state of the query-network running at a node. This state is captured by: the internal state of each operator, the contents of operator's input queues, and the contents of a node's output queues. To avoid losing any tuples that are in-flight between two nodes when a failure occurs, output queues should preserve tuples until they have been checkpointed and acknowledged by the downstream nodes. Furthermore, to reduce the size of checkpoint messages, downstream nodes can acknowledge tuples only *after* they have been processed by the first set of operators, thus completely avoiding checkpointing the contents of their input queues (except for the identifiers of the latest acknowledged tuples).

The frequency of checkpoint messages, specified by the *checkpoint interval*, determines the tradeoff between runtime overhead and recovery time. Frequent checkpoints shorten recovery time since less processing needs to be repeated but increase runtime overhead by increasing both the bandwidth used to transmit checkpoint messages and the processing required to create them.

When a primary fails, the backup takes over and sends all tuples from its output queues to the primary's downstream neighbors. Also, the failed primary's upstream neighbors start sending tuples (including those in their output queues) to the backup, which starts processing them starting from the latest acknowledged tuples. When the old primary comes back up, it becomes the new secondary.

4.2 Upstream Backup

Given the significant overhead of process-pair-based approaches, we propose an alternate model that leverages the network-flow form of processing to reduce runtime overhead at the expense of a small increase in recovery time.

In this approach, *upstream nodes act as backups for their downstream neighbors*. For instance, in the network shown on Figure 2, nodes N_i and N_k serve as backups for node N_j. To serve as backup, each node holds tuples in its output queues until all its downstream neighbors have completely processed them. By doing so, if a downstream node fails, any other node can restore the lost state by replaying the tuples logged by the upstream neighbors. Tuples are trimmed (i.e., removed) from output queues once they have been fully processed by all downstream neighbors *and* tuples resulting from that processing have safely been transferred further down the network. In Figure 2, nodes N_i and N_k hold tuples in their output queues until N_j and the other downstream neighbors have all safely processed the tuples and have forwarded resulting tuples to the application. If N_j fails, another node, the failover node, can recreate N_j's state by re-processing the tuples logged by N_i and N_k.

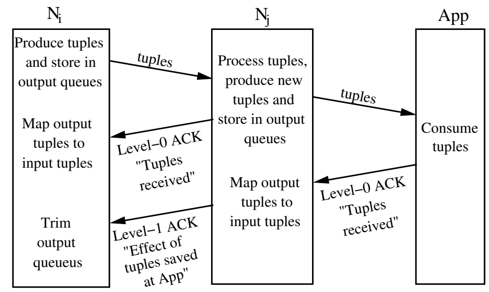


Figure 3. Overview of inter-node communication in upstream backup

We assume that the description of the network of operators running at each node is stored in a distributed catalog and the failover node uses the information from that catalog to re-construct the missing network of operators.

With this approach, downstream nodes must periodically inform their upstream neighbors that it is safe to discard some logged tuples. We call such notifications *queue-trimming messages* since their reception results in an upstream node trimming its output queue. The difficulty of generating these notifications is twofold. First, for the approach to scale and provide fault tolerance in the face of multiple concurrent failures (as discussed in Section 4.4), all notifications must be exchanged only between immediate neighbors. Second, nodes must be able to map every tuple they produced to the earliest input tuples that were used to generate these outputs tuples (as discussed in Section 4.2.2).

Figure 3 shows three nodes (N_i, N_j, and App) and a typical communication sequence. When a node produces tuples, it transmits them to its downstream neighbor and stores them in its output queue. Periodically, each node acknowledges the tuples it receives by sending a level-0 acknowledgment to its immediate upstream neighbor. The reception of these messages indicates to a node that some of the tuples it produced have now been saved at the downstream neighbor. The node, N_j for instance, can then inform its upstream neighbor (N_i), by sending a level-1 acknowledgment, that the tuples that contributed to producing the acknowledged tuples can now be discarded from the upstream neighbor's (N_i's) output queue. Each node trims its output queue when it receives a level-1 acknowledgment (i.e., a queue-trimming message). Leaf nodes in the query-processing graph, such as N_j in Figures 2 and 3, use level-0 acknowledgments to trim their output queues.

4.2.1 Queue Trimming Protocol

We first describe the details of queue-trimming message generation and output-queue truncation assuming the exis-

01. repeat forever
02. wait to receive an $ACK(0, S_o, v)$ from N_i
03. $down\_bound[0, S_o, N_i] \leftarrow v$
04. $w \leftarrow min\{down\_bound[0, S_o, N_i] : N_i \in N\}$
05. if $w > min\_downbound[0, S_o]$
06. $min\_downbound[0, S_o] \leftarrow w$
07. $\forall S_i \in S_{input}$ such that $\exists$ a path from S_i to S_o
08. $up\_bound[0, S_i, S_o] \leftarrow cause((S_o, w), S_i)$
09. $x \leftarrow min\{up\_bound[0, S_i, S_o] : S_o \in S_{output}\}$
10. if $x > min\_upbound[0, S_i]$
11. $min\_upbound[0, S_i] \leftarrow x$
12. acknowledge $ACK(1, S_i, x)$

Figure 4. Algorithm for generating level-1 acknowledgments

repeat forever
wait to receive an $ACK(1, S_o, v)$
$down\_bound[1, N_i, S_o] = v$
$w \leftarrow min\{down\_bound[1, N_i, S_o] : N_i \in N\}\}$
if $w > min\_downbound[1, S_o]$
$min\_downbound[1, S_o] \leftarrow w$
trim output queue for S_o
up to but excluding $min\_downbound[1, S_o]$

Figure 5. Queue-trimming algorithm performed by each node

tence of a function that determines, for a given node, which input tuples caused the generation of some output tuples. We define this function as $cause((S_o, v), S_i) \rightarrow (S_i, u)$, where S_i and S_o identify streams, and u and v identify tuples on S_i and S_o respectively. Given an output tuple identifier (S_o, v) and an input stream identifier S_i, this function returns the identifier, (S_i, u) that marks the beginning of the sequence of tuples on S_i necessary to regenerate (S_o, v). We defer the discussion of this function to Section 4.2.2.

To avoid possible loss of tuples in-flight between neighbors, nodes generate queue-trimming messages only after tuples have safely been received by all downstream neighbors. Level-0 acknowledgments, denoted with $ACK(0, S_i, u)$, are used to confirm tuple reception at the application level. Each node produces an acknowledgment for every M_n tuples it successfully receives, every M_s seconds, or whichever occurs first. Intuitively, a lower acknowledgment frequency reduces bandwidth utilization overhead, but increases the size of output queues and the recovery time. We analyze this tradeoff in Section 5.

When a node receives level-0 acknowledgments from all its downstream neighbors on all its output streams, it notifies its upstream neighbors that the effects of processing some of their tuples have been saved at all downstream neighbors. These notifications are called level-1 acknowledgments (denoted $ACK(1, S_i, u)$), since using them to trim queues allows the system to tolerate any single failure.

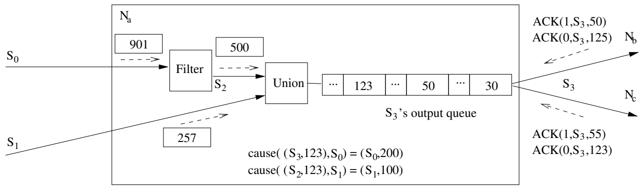
Figure 4 shows the pseudocode of the algorithm for generating level-1 acknowledgments. Basically, every time *all* downstream neighbors on a stream S_o acknowledge a tuple, the node maps that tuple back up onto the earliest input tuple, on each input stream, that "caused" it. For each input stream, the node identifies the most recent tuple whose effect has propagated in such a manner *on all output streams*. The node produces a level-1 acknowledgment for that tuple.

In this figure, we denote with N the set of downstream neighbors of a node, and with S_{input} and S_{output} the sets of all input and output streams respectively. We also denote with $S_i[u]$, tuple u on stream S_i and with (S_i, u), the identifier of that tuple. Each node uses map $down\_bound[0, S_o, N_i]$ to remember, for each output stream, the latest tuple acknowledged at level-0 by each downstream neighbor, and $min\_downbound[0, S_o]$ to remember the latest tuple acknowledged by all these neighbors. Similarly, to keep track of input tuples for which queue-trimming messages can be issued, each node uses map $up\_bound[0, S_i, S_o]$ and $min\_upbound[0, S_i]$.

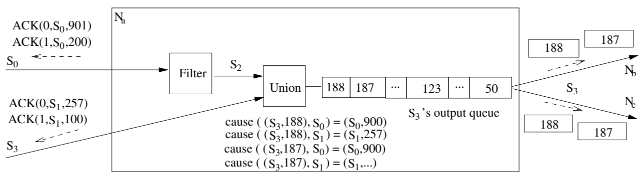
The detailed algorithm for generating level-1 acknowledgments in Figure 4 proceeds as follows. When a node receives a level-0 acknowledgment from a neighbor N_i for a stream S_o, it updates the value of the last tuple acknowledged by that neighbor on that stream (line `02-03`). The node recomputes the minimum value acknowledged across all neighbors for that stream (line `04`). If the value has increased (lines `05-06`), the acknowledged tuple is mapped onto each input stream S_i that contributes to tuples of S_o. The values resulting from these mappings are written in up_bound (lines `07-08`). The actual tuple that determines the new trimming bound on each stream S_i is the minimum of all values in up_bound, associated with S_i (lines `09-11`). The node generates a level-1 acknowledgment (line `12`) for that tuple (as a direct optimization, nodes can also generate level-1 acknowledgments periodically).

As the upstream neighbor receives level-1 acknowledgments, it trims its output queues up to the maximum values acknowledged at level-1 by all its downstream neighbors. Figure 5 shows the detailed algorithm.

Figure 6 illustrates one iteration of the upstream-backup algorithms on one node. Figure 6(a) shows node N_a receiving level-0 and level-1 acknowledgments from two downstream neighbors N_b and N_c: $ACK(0, S_3, 125)$, $ACK(0, S_3, 123)$, $ACK(1, S_3, 50)$, and $ACK(1, S_3, 55)$. Tuple $S_3[50]$ has thus been acknowledged by both neighbors at level-1 and can be used to trim the output queue (Figure 6(b)). Tuple $S_3[123]$ has now been acknowledged by both downstream neighbors at level-0. $S_3[123]$ maps onto input tuples identified with $(S_0, 200)$ and $(S_1, 100)$ which can thus be acknowledged at level-1. While processing acknowledgments, the node receives tuples $S_0[901]$ and $S_1[257]$ from its upstream neighbors. The node can ac-



(a) A node receives acknowledgments from its downstream neighbors, while receiving new tuples from its upstream neighbors. In the figure, the filter operator also produces a new tuple.



(b) The node maps the lowest level-0 acknowledgment received, $(S_3, 123)$, onto level-1 acknowledgments on all its input streams. The node trims its output queue at the lowest level-1 acknowledgment received: $(S_3, 50)$. The node also places new tuples produced in its output queue.

Figure 6. One iteration of upstream-backup

knowledge these tuples at level-0, assuming $M_n = 1$.

4.2.2 Mapping Output Tuples to Input Tuples

To generate appropriate level-1 acknowledgments from level-0 acknowledgments, a node must map arbitrary output tuples to the earliest input tuples that contributed to their production, i.e., the node must be able to compute $(S_i, u) \leftarrow cause((S_o, v), S_i)$. Such a mapping is operator dependent. For simple operators such as filters or maps, each output tuple corresponds directly to an input tuple. For operators that manipulate windows of tuples or preserve some temporary state, the same approach can also be applied but only if the operators are able to identify, at any given time, the first tuple on each of their input streams required to regenerate their complete current state. For a query-network, the $cause$ function must yield the *oldest* (based on a locally assigned sequence number) input tuple that contributes to the state of any operator in the network.

The mapping is facilitated by appending *queue-trimming offset-indicators* to tuples as they travel through operators on a node. For any tuple (S, w) (on an intermediate stream or an output stream), these indicators, denoted with $indicators((S, w))$, represent the set of input tuples, (S_i, u), that cause the production of (S, w): each (S_i, u) is the oldest input tuple on S_i necessary to generate $S[w]$.

When a tuple $S_i[u]$ enters a node, its indicators are thus initialized to the tuple itself. Figure 7 presents the algorithm

1. At each operator box b
2. for each output tuple produced $b.S_o[v]$
3. $\forall b.S_i \in b.S_{input}$
4. get oldest state-contributing tuple $(b.S_i, u)$
5. if $indicators(b.S_i, u) = \emptyset$
6. $indicators(b.S_i, u) \leftarrow \{(b.S_i, u)\}$
7. $\forall i \in indicators(b.S_i, u)$
8. $indicators(b.S_o, v) \leftarrow indicators(b.S_o, v) \cup \{i\}$

Figure 7. Algorithm for setting queue-trimming offset-indicators on output tuples, at each box

for appending and composing indicators. When an operator b produces a tuple, it determines the oldest tuple in its input queue that has contributed to the state yielding the production of the new tuple (lines 02−04). The operator reads the indicators from the input tuple and appends them to the indicators of the output tuple (lines 07−08). The first operators to see tuples initialize their indicators (lines 05−06). The last operators to see tuples before they are forwarded to downstream neighbors (the output operators) keep indicators in output queues, but they do not send them further (not shown). Operators with multiple inputs append multiple indicators to output tuples, one indicator per input that *actually contributed to the generation of the tuple*. For instance, a union operator [1] would only append the indicator for the stream from which the output tuple was issued.

Applying the *cause* function thus consists in reading the set of indicators of a tuple in the output queue: i.e., $cause((S_o, v), S_i) = \{(S_x, y) \in indicators((S_o, v)) : x = i\}$. Due to operators such as union, however it is possible that there exists a path from S_i to S_o but $cause((S_o, v), S_i))$ returns an empty set. In that case, the value used should be the first non-empty set returned by the $cause$ function for the tuples preceding u on S_o.

Operators with low selectivity may unnecessarily delay queue-trimming by producing offset-indicators at a coarse granularity. To avoid this problem, these operators may introduce *flow tuples* (i.e., null output tuples) which are forwarded unchanged through all the operators and only serve the purpose of reducing the granularity of queue trimming. Flow tuples are not forwarded to downstream neighbors.

Figure 6 shows an example of executing the offset-indicator management algorithm. In Figure 6(a), the filter operator produces tuple $S_2[500]$ which corresponds to the previously received input tuple $S_0[900]$. Hence, $indicators((S_2, 500)) = \{(S_0, 900)\}$. In Figure 6(b), the union operator processes tuples $S_2[500]$ and $S_1[257]$ to produce $S_3[187]$ and $S_3[188]$ respectively. These tuples are forwarded to downstream neighbors and placed in the output queue. Moreover, $indicators((S_3, 187)) = \{(S_0, 900)\}$ and $indicators((S_3, 188)) = \{(S_1, 257)\}$. Therefore, $cause((S_3, 188), S_0) = (S_0, 900)$, $cause((S_3, 188), S_1) = (S_1, 257)$, and $cause((S_3, 187), S_0) = (S_0, 900)$.

$cause((S_3, 187), S_1)$ depends on the indicators on the tuples preceding S_3.

4.3 Active Standby

Finally, the last approach that we study is a variation of process pairs but uses some features of upstream backup. We call this approach active standby. Similarly to passive standby, each processing node has a dedicated secondary node. Unlike passive standby, however, secondary nodes actively obtain tuples from their primaries' upstream neighbors and process them in parallel with their primaries.

The main advantage of active standby is its faster recovery time compared with passive standby. In active standby, when a backup node takes over, it does not need to repeat all processing since the last checkpoint, because it kept processing tuples even when the primary was down. This advantage, however, comes at the cost of using extra bandwidth for sending input tuples to the secondary and using extra processing power for duplicating all processing.

In active standby, secondary nodes log the tuples they produce in output queues without sending them. These tuples are discarded only when the primary node delivers all the corresponding tuples to its downstream neighbors. If the primary fails, the secondary takes over and re-sends the logged tuples to all downstream neighbors.

Since the primary and secondary may schedule operators in a different order, they might have different tuples in their output queues. This makes the task of bounding output queues at the secondary particularly difficult. We tackle this problem by exploiting queue-trimming offset-indicators introduced in Section 4.2.2. When a primary receives a level-0 acknowledgement from all its downstream neighbors on a stream, it extracts the offset-indicators from the acknowledged tuples and uses them to compose a queue-trimming message *for the secondary node*. When the secondary receives that message, it retrieves the trimming bound for each output queue and discards the appropriate tuples.

We use the example from Figure 6 to illustrate active standby. When $ACK(0, S_3, 125)$ and $ACK(0, S_3, 123)$ arrive, node N_a determines that $S_3[123]$ is now acknowledged at level-0 by both downstream neighbors. Since tuple $S_3[123]$ maps onto input tuples identified with $(S_0, 200)$ and $(S_1, 100)$, the set of identifiers $\{(S_0, 200), (S_1, 100)\}$ is added to the queue-trimming message as the entry value for S_3. When the secondary receives the queue-trimming message, it discards tuples u (from the output queue corresponding to S3) for which both $cause((S_3, u), S_0)$ returns a tuple older than $(S_0, 200)$ and $cause((S_3, u), S_1)$ returns a tuple older than $(S_1, 100)$.

4.4 K-safety

Tolerance to single failures significantly increases the availability of a system. However, circumstances may arise where multiple nodes fail concurrently (an error in a recent upgrade, a large-scale natural catastrophe, or even an unlikely coincidence). All approaches described above can be extended to provide a higher degree of fault tolerance, allowing the system to survive concurrent failures of up to K nodes. We call such resilience *K-safety*.

To achieve K-safety with passive or active standby, K backup nodes must be associated with each primary node. Each backup must receive checkpoint messages or must process tuples in parallel with the primary. To extend the upstream-backup approach to K-safety, each node must preserve tuples in its output queues until they have been processed by K consecutive nodes on all K-length downstream paths. To do so, level-1 acknowledgments must be used to generate level-2 acknowledgments, and so on until a node receives a level-K acknowledgment. Only level-K acknowledgments are actually used to trim output queues.

Even with some optimizations, the overhead of high availability grows linearly with K for all approaches: the number of checkpoint messages or the number of acknowledgments is proportional to K. For the upstream-backup model, however, the rate of the increase is much lower since the basic single-fault-tolerance overhead is low. This approach is thus better suited to achieve higher fault tolerance.

4.5 Duplicate Tuples

All three approaches may generate duplicate tuples during failure recovery. With passive standby, when the secondary takes over, it resends all tuples not yet acknowledged at the moment of the last checkpoint. Since the primary could have failed after sending these tuples, they might be received twice by downstream nodes. Duplicates, however, are trivial to eliminate using the tuple identifiers, because the tuples are exactly the same as the original ones.

For upstream backup, duplicate elimination is somewhat more involved. When a failure occurs, the backup node reprocesses all tuples logged in the output queues and produces new tuples. These new tuples have the same content as the original ones, but they may appear in a different order. Duplicate elimination is therefore most easily performed by the client itself in an application-specific manner. We do not address duplicate elimination in this paper.

Active standby also does not guarantee that primary and secondary nodes are always in exactly the same state. Although both the primary and secondary nodes process tuples in parallel, they may schedule operators in a different order. During recovery, the active-standby approach may therefore generate duplicate tuples, as in the case of upstream backup.

5 Evaluation

In this section, we use analysis and simulation to evaluate and compare the basic runtime overhead and recovery-time performance of the three high-availability approaches.

Parameter	Meaning	Value
Stream Rate (tuples/sec)	# of tuples generated by a stream source per sec.	500
Tuple Processing Cost (microsec.)	Time for an Aurora box to process one tuple	10
Tuple Size (bytes)	Size of a tuple	50
Tuple Identifier Size (bytes)	Size of a tuple identifier	8
Selectivity	For each box the expected value of $\frac{\text{\# of output tuples emitted}}{\text{\# of input tuples consumed}}$	variable
Network Delay (millisec.)	Network link latency between any pair of nodes	5
Network Bandwidth (Mbps)	Bandwidth between any pair of nodes	8
Failure Detection Delay (millsec.)	Delay to detect the failure of a node	100
Failure Duration (sec.)	Time for a failed node to recover	100
Queue-Trimming Interval (millisec.)	Time interval between consecutive queue-trimming messages (upstream backup, eager standby)	variable
Checkpoint Interval (millisec.)	Time interval between consecutive checkpoint messages (process-pair)	variable

Table 1. Simulation parameters

Measure	Meaning
Processing Delay (millisec.)	Time difference between the moment an output tuple t was produced and the moment the earliest tuple that contributed to generating t was produced.
Partial Recovery Time (millisec.)	Time interval between the detection of a failure and the reception of the first non-duplicate tuple by a node (or application) downstream from the failure.
Full Recovery Time (millisec.)	Time interval between the detection of a failure and the moment when the tuple end-to-end processing delay returns to the pre-failure value.
Relative HA Overhead	$\frac{\text{total data transferred for high availability}}{\text{total data transferred for tuple processing}}$

Table 2. Measured statistics

5.1 Simulation Environment

We built a detailed simulator modeling a distributed stream-processing environment. The simulator, implemented using CSIM [16], models (1) stream sources that continuously generate tuples and inject them into the system, (2) processing nodes on which operators, specified as part of query networks, process tuples and produce new tuples, and (3) client applications that act as tuple sinks. Each experiment was run for five simulation minutes for warm-up and another 10 simulation minutes for statistics collection. Unless indicated otherwise, each point shown in a figure is an average result of 25 independent simulation runs. Tables 1 and 2 summarize the simulation parameters and the measured statistics that we use as metrics, respectively.

5.2 Basic Runtime Overhead

We first compare the overhead of the approaches using a simple scenario, where the system consists of only two nodes, one upstream from the other. Each node runs a single operator with a single output stream. Table 3 summarizes the parameter values used for this micro-benchmark. We measure the overhead during normal operation.

Parameter	Values	Default
Selectivity	0.2, 0.4, 0.6, 0.8, 1.0	1.0
Q. Tr. Interval (millisec.)	10, 20, 40, 60, 80, 100	10
Checkpoint Interval (millisec.)	10, 20, 40, 60, 80, 100	10

Table 3. Parameters values used in simulations

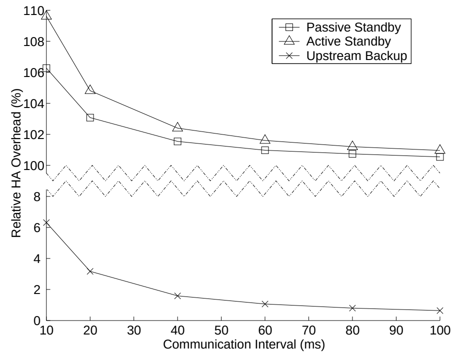


Figure 8. High-availability overhead for various queue-trimming or checkpoint intervals. The 8% to 100% range on the y-axis is removed for clarity.

The main component of overhead is the bandwidth required to transmit checkpoint or queue-trimming messages. Output queues also contribute to overhead, but since queue sizes and bandwidth utilization are inversely proportional and are both determined by queue-trimming intervals, we focus our overhead analysis on bandwidth utilization only. Longer output queues also translate into longer recovery times which we analyze in the following section.

Figure 8 shows the relative high-availability overhead for each approach when all parameters have their default values and selectivity is 1.0. The y-axis represents the overhead computed as the ratio (%) of bandwidth used for high availability purposes over the bandwidth used without high availability. Since the overhead increases with increasing checkpoint or queue-trimming-message frequency, we compute the overhead for various *communication intervals* (x-axis) between such messages. The figure shows that passive and active standby incur almost the same overhead (100% to 110%), which is significantly greater than that of upstream backup (less than 10%).

These results can be explained as follows. Since the node runs a single operator, for passive standby each checkpoint message contains the set of tuples produced in the last checkpoint interval M_s as well as the identifier of the last tuple acknowledged by all downstream neighbors (i.e., the head position in the output queue). Therefore, all tuples produced are transferred to the backup creating a 100% overhead. The additional overhead is due to the transmission of the head positions (from primary to secondary) and input-tuple acknowledgements (from primary to upstream node). Both are generated every M_s seconds. Assuming that x is the size of tuple identifiers used for head positions and acknowledgements, the total overhead is then:

$$B_{passive\_standby} = \frac{\lambda c M_s + 2x}{\lambda c M_s} = 1 + \frac{2x}{\lambda c M_s} = 1 + \frac{\alpha}{\lambda M_s} \quad (1)$$

where c is the tuple-size and λ is the tuple arrival-rate. $\lambda c M_s$ is the byte-wise amount of tuples received and produced during the time-interval M_s. A shorter checkpoint interval, thus increases overhead by sending queue head-positions and acknowledgements more frequently.

In active standby, *input* tuples sent to the primary are also sent to the backup, causing a 100% overhead in bandwidth utilization. Additionally, every M_s sec a queue-trimming message is sent from primary to secondary and acknowledgements are sent from both primary and secondary to the primary's upstream neighbor. Active-standby's overhead is thus just a little over that of passive standby for the scenario simulated (one extra x-byte identifier sent every M_s sec).

In contrast, in upstream backup, only one level-0 and one level-1 acknowledgement, per stream crossing a node boundary, are sent every M_s seconds. The overhead is thus:

$$B_{upstream\_backup} = \frac{2x}{\lambda c M_s} = \frac{\alpha}{\lambda M_s} \quad (2)$$

The key advantage of upstream backup is therefore to avoid sending duplicate tuples to backup nodes.

5.3 Basic Recovery Time

We now consider the time to recover from single failures. We use the same network configuration as above: one node upstream of one other, each node running one operator. We simulate the failure of the downstream node and the recovery of the failed piece of Aurora network on a third spare node.[1]

We analyze the end-to-end tuple delay during failure and recovery and measure the recovery times (as defined in Table 2) of each approach. Figures 9 and 10 present the results obtained. In Figure 9, each point represents a tuple: the x-coordinate indicates the time when the application receives the tuple, with the failure occurring at time 0 sec. The y-coordinate represents the end-to-end processing delay of the tuple. Using active standby as illustration, the figure also summarizes the main parameters of recovery. When a failure occurs, it takes D msec before the failure is detected (100 msec in the simulation). Once recovery starts, it takes P msec, the partial recovery time, for the application to receive the first non-duplicate tuple. It takes R msec for the end-to-end tuple delay to go back to its value before failure. At that point, the recovery is fully completed.

[1] In passive standby and active standby, when a node fails, its query-network is recovered on a pre-assigned backup node. In upstream backup, recovery can be performed on any node. For fairness, we choose to recover on a spare node as well.

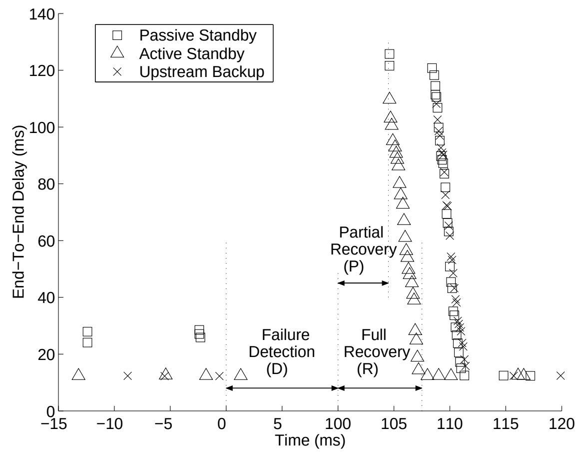


Figure 9. Dynamics of failure-recovery. The time interval between 0 msec and 100 msec on the x-axis is shortened for clarity.

The duration of partial recovery is determined by three factors: (1) the time to re-create the query network of the failed node, (2) the delay, K, before the first recovered tuple reaches the node downstream from the failure, and (3) the amount of time it takes to re-generate and re-transmit all duplicate tuples. For (1), we assume that the time is included in the failure detection delay. For (3), we assume that tuple processing is much faster than tuple transmission. The partial recovery time is thus given by:

$$P = K + \frac{Qc}{B} \quad (3)$$

where Q is the number of duplicate tuples; c is the tuple size; and B is the network bandwidth.

For upstream backup, K is the sum of (i) the transmission delay from upstream node to recovery node, (ii) the processing delay at the recovery node, and (iii) the transmission delay from recovery node to destination. For active standby, K is only (iii), because the backup node keeps processing tuples while the failure is being detected. For passive standby, K can be either the same as active standby or the same as upstream backup, depending whether the primary failed before or after transmitting its output tuples.

For upstream backup and active standby, Q is determined by the queue-trimming interval and by network delays. Since level-0 acknowledgements are generated every M_s seconds, there are on average $\frac{M_s \lambda}{2}$ un-acknowledged tuples. Even for $M_s = 0$ there is a constant delay between tuple production and the reception of their level-0 acknowledgements. With a d sec network delay, a λ tuples/sec arrival rate, and assuming level-1 acknowledgements are triggered by level-0 acknowledgements, the average number of duplicate tuples in output queues is thus:

$$Q = \frac{M_s \lambda}{2} + 2d\lambda \quad (4)$$

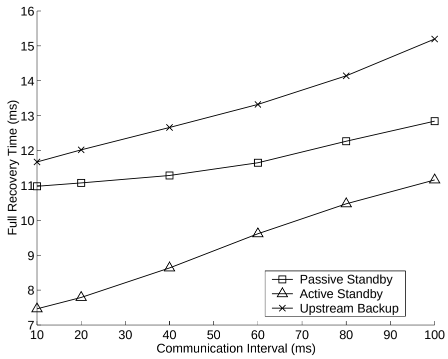


Figure 10. Impact of increasing communication interval on full recovery time

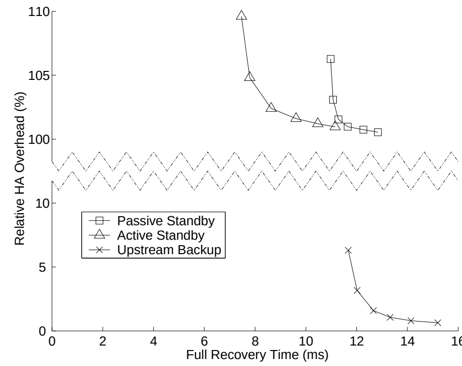


Figure 11. Tradeoffs between relative HA overhead and full recovery time

For passive standby, if the primary fails after sending its output tuples, all tuples re-processed since the last checkpoint are duplicates. Hence, Q is approximately $\frac{M_s\lambda}{2}$ on average.

The results confirm that active standby has the shortest partial recovery time ($\approx$ 4 msec) and upstream backup has the longest ($\approx$ 9 msec). The results presented in that figure show exactly one run. In that experiment, when the failure occurred, the output queue of the backup node in passive standby contained non-duplicate tuples making partial recovery the same as that of active standby.

Once a failure is detected, the *full recovery time* is the amount of time it takes for the output-queue draining process to catch up with the queue-filing process. Roughly, the output queues are drained at network transmission speed ($\frac{B}{c}$ tuples/sec) and are filled up at λ tuples/sec. The difference between the two values determines the slope of the curve. Hence, the full recovery time is determined by the detection delay (and whether tuples are processed or accumulated during that time), the partial recovery time, and the slope of the recovery curve.

Figure 10 shows the full recovery time (y-axis) of each approach for various queue-trimming or checkpoint intervals (x-axis). As the communication interval increases, Q increases, lengthening the partial recovery time P and thus the full recovery time. More importantly, when the communication interval is doubled, the runtime overhead of upstream backup is halved (Figure 8), but its recovery time (Figure 10) increases only linearly. For instance, when M_s increases from 40 msec to 80 msec, the overhead drops from 2% to 1% while the recovery time increases only from 12.5 msec to 14 msec). For the other approaches, recovery time also increases linearly with the communication interval, but the overhead remains roughly constant above 100%.

Finally, Figure 11 summarizes the tradeoffs between runtime overhead and recovery time. Each point on a curve corresponds to a different communication interval. The figure shows that, compared with active standby, upstream backup increases failure-recovery time approximately from 8 msec to 12 msec for a query network consisting of one operator. The increase is even smaller compared with passive standby (11 msec to 12 msec only). If we include the failure detection delays, which are significantly longer, the added recovery time becomes negligible: the total recovery increases from 108 to 112 msec. On the other hand, upstream backup reduces runtime overhead by approximately 95% compared with both approaches. It is therefore the only approach that makes it possible to achieve good recovery-time performance with a relatively small overhead.

# boxes (1.0 selectivity)	1	4	9	16	25
overhead (%)	106.3	109.4	112.4	115.7	118.7
selectivity (25 boxes)	1.0	0.8	0.6	0.4	0.2
overhead (%)	118.5	98.8	78.3	57.1	33.2

Table 4. Impact of query-network complexity on passive-standby overhead. The overheads of active standby and upstream backup remain constant at 110% and 6% respectively

5.4 Effects of Query-Network Size and Selectivity

Table 4 shows the impact of varying the number of operators and operator selectivity on the overhead of passive standby. The overhead of the other approaches is independent of these factors. Boxes had a fanout of 1 (i.e, each box had a single output arc) and were placed such that the query network had roughly the same width and depth. As the number of boxes increases, passive standby consumes more bandwidth and quickly exceeds the overhead of active standby (crossing point is at four operators in the simulation). Indeed, in passive standby, each checkpoint message includes the content of *each* operator's input queue and the node output queues. In active standby, in contrast, only duplicate *input* tuples are sent to the backup.

Unlike the other approaches, passive standby is able to exploit operator selectivity, incurring lower runtime overhead with decreasing selectivity. Since passive standby does

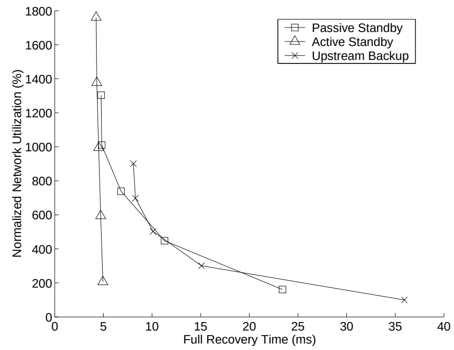


Figure 12. Tradeoffs between distribution overhead and recovery time

not include input queues in checkpoint messages, tuples dropped due to selectivity never appear in checkpoints, thus lowering the overhead.

Independently of configuration, though, the overhead of upstream backup is significantly lower than those of the other two approaches.

5.5 Effects of Query-Network Distribution

Finally, Figure 12 shows the increase in bandwidth utilization (y-axis) as a query network of 100 boxes is spread across increasing numbers of primary nodes (4, 16, 36, 64, and 100). The overhead is normalized by the bandwidth used by upstream backup with 4 nodes. The figure also shows how distribution decreases recovery time (x-axis), since any failure requires recovering a query network with fewer operators and fewer input queues. We recovered failed query-networks on spare backup nodes for active standby and passive standby, but on randomly chosen live nodes for upstream backup, making the comparison unfavorable to the latter (resulting in slightly longer recovery times).

Upstream backup gains most from distribution: smallest bandwidth utilization increase (maximum 900% *vs* 1800% for active standby) for largest recovery time decrease (36 msec down to 7 msec). Indeed, for the process-pair based approaches, any additional node requires up to twice as much bandwidth as one additional node in upstream backup. Active standby does not benefit from distribution at all since recovery does not involve re-processing. For the other two approaches, distribution helps most when the load is high: when going from 4 to 16 nodes, full recovery is reduced from 36 msec to 15 msec for upstream backup and from 23 msec to 12 msec for passive standby.

In conclusion, the evaluation shows that active standby incurs by far the highest overhead of all the approaches, but at the benefit of short recovery times. Passive standby incurs a somewhat lower overhead, but the overhead of that approach is more difficult to predict as the effects of number of operators and selectivities on overhead are opposite. Upstream backup is by far the lightest of all three approaches, in all configurations. Its recovery-time performance is worse than that of active standby but similar to that of passive standby. In all cases, recovery is however dominated by failure detection. In our experiments, actual recovery durations are in the order of 10s msec whereas failure detection is in the order of 100s msec. Increasing detection and recovery from approximately 105 msec to 115 msec makes the recovery overhead incurred by upstream backup almost negligible.

For each approach, the tradeoff between overhead and recovery time can be somewhat adjusted by varying the communication intervals between nodes. Since overhead is inversely proportional to the communication interval whereas recovery time is directly proportional to it, a value at the knee of the overhead/recovery-time curve is a good tradeoff.

6 Related Work

Recently, there has been much work on data-stream processing (e.g., [1, 4, 5, 18, 20]). In addition to Aurora, which we discuss in Section 2, NiagaraCQ [6], STREAM [18] and TelegraphCQ [5] are also building general purpose stream processing engines. Although we discuss our approaches in the context of Aurora, our basic findings are general.

More recently, there have been proposals that extend single-server stream processing systems to distributed models and environments. The fundamental idea behind Aurora* and Medusa [7] is to transparently partition the logical Aurora processing network into multiple sub-networks and execute them on distributed machines to increase system scalability, robustness, and performance.

This paper discusses approaches for enhancing the availability of DSPSs such as Aurora* or Medusa. High availability has traditionally been achieved using two basic techniques. In the first approach, often called process pairs, multiple copies of the same data are stored on disks attached to separate processors. When one copy fails, another takes overs. Examples of this approach include mirrored disks (e.g. Tandem) [21], interleaved de-clustering, and chained de-clustering [11]. In the second approach, which is not directly applicable to a DSPS, data and corresponding error detection/correction information are spread across an array of disks [10]. When errors are discovered, this redundant information is used to restore the data.

Similarly to other data management systems [8, 17, 19], many commercially available workflow systems [12] rely on redundant components to achieve high availability. A variation of the process-pairs approach is used in the Exotica workflow system [13]. Instead of backing up process states, Exotica logs changes to the workflow components, which store inter-process messages. This approach is similar to upstream backup in that the system state can be recovered by reprocessing the component backups. Unlike upstream backup, however, this approach does not take advantage of the data-flow nature of processing, and therefore has to explicitly back up the components at remote servers.

The DR scheme [14], which efficiently resumes failed warehouse loads, is similar to upstream backup. Instead of offset-indicators, DR uses output tuples and properties of operators to compute, during recovery, the trimming bounds on input streams. In contrast to DR, our scheme supports infinite inputs by trimming output queues at runtime. We also support failure recovery at the granularity of nodes instead of the whole system. We do not require that input streams have any property such as order on some attribute.

7 Conclusions

Many stream-oriented applications are mission critical and demand high availability. Previous work in stream-processing has focused on performance and language issues, ignoring equally important issues that revolve around achieving high availability. This paper presents an initial step toward high availability in stream-processing systems.

We have argued that the streaming characteristics of our target systems and applications raise a number of interesting challenges and opportunities. We discussed how traditional availability approaches can be adapted to distributed stream-processing environments. We also described a new stream-oriented approach that leverages the data-flow nature of distributed stream processing.

We compared our approaches using analysis and simulation across a variety of configurations. Our results quantitatively characterized the runtime vs. recovery-time overheads of the approaches. In particular, the results revealed that while traditional approaches can be effectively used to achieve low recovery times, exploiting the natural flow of data can significantly reduce runtime overheads (at the expense of a small increase in recovery times). We believe that, given the continuous and network-centric nature of stream-processing applications and the ever-improving hardware reliability factors, protocols that optimize for runtime messaging overhead will be increasingly more desirable.

There are several immediate directions for future research. First, we plan to provide support for more complex operators that include windows and persistent storage. Second, we plan to address the choice of failover nodes, which is a non-trivial problem as solutions need to take into account potentially conflicting goals such as balancing load and preserving query locality. Finally, we plan to implement and deploy our approaches, in the context of Aurora* and Medusa, to verify their practicality and effectiveness.

8 Acknowledgments

We would like to thank the members of Aurora and Medusa projects for many valuable discussions and feedback. We also acknowledge Hiro Iwashima for his help on early versions of the simulator.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal: The International Journal on Very Large Data Bases*, to appear, sep 2003.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of 2002 ACM Symposium on Principles of Database Systems (PODS 2002)*, June 2002.

[3] J. Barlett, J. Gray, and B. Horst. Fault tolerance in Tandem computer sytems. Technical Report 86.2, Tandem Computers, Mar. 1986.

[4] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Aug. 2002.

[5] S. Chandrasekaran, A. Deshpande, M. Franklin, and J. Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data*, May 2000.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[8] E. Cialini and J. Macdonald. Creating hot snapshots and standby databases with IBM DB2 Universal Database$^{(TM)}$ V7.2 and EMC TimeFinder$^{(TM)}$. DB2 Information Management White Papers, Sept. 2001.

[9] J. Gray. Why do computers stop and what can be done about it? Technical Report 85.7, Tandem Computers, June 1985.

[10] J. Gray, B. Horst, and M. Walker. Parity striping of disc arrays: Low-cost reliable storage with acceptable throughput. In *Proc. of the 16 International Conference on Very Large Database*, 1990.

[11] H.-I. Hsiao and D. DeWitt. Chained Declustering: A new availability strategy for multiprocessor database machines. In *Proc. of 6th International Data Engineering Conference*, pages 456–465, 1990.

[12] IBM Corporation. IBM WebSphere V5.0: Performance, scalability, and high availability: WebSphere Handbook Series. IBM Redbook, July 2003.

[13] M. Kamath, G. Alonso, R. Guenthor, and C. Mohan. Providing high availability in very large workflow management systems. In *Proc. of 5th International Conference on Extending Database Technology, Avignon, March 1996.*, 1996.

[14] W. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik. Efficient resumption of interrupted warehouse loads. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data*, May 2000.

[15] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of the 18th International Conference on Data Engineering*, 2002.

[16] Mesquite Software, Inc. CSIM 18 user guide. `http://www.mesquite.com`.

[17] Microsoft Corporation. Microsoft SQL server 2000 high availability series. Microsoft TechNet, 2003.

[18] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the First*

Biennial Conference on Innovative Data Systems Research (CIDR'03), Jan. 2003.

[19] A. Ray. Oracle data guard: Ensuring disaster recovery for the enterprise. An Oracle white paper, Mar. 2002.

[20] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of the 19th International Conference on Data Engineering (ICDE 2003)*, Mar. 2003.

[21] Tandem Database Group. Non-stop SQL: A distributed, high performance, high-reliability implementation of SQL. In *Proc. of the Workshop on High Performance Transaction Systems*, 1987.