High-Availability Algorithms for Distributed Stream Processing

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

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-04-05
May 2004

High-Availability Algorithms for Distributed Stream Processing*

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

Stream-processing systems are designed to support an emerging class of applications that require sophisticated and timely processing of high-volume data streams, often originating in a distributed environment. Work in stream processing has so far focused primarily on stream-oriented languages and resource-constrained, one-pass query processing. High availability, a key goal for virtually all data-processing systems, has received little attention until now.

This paper investigates failure-recovery approaches for distributed stream-processing systems. In these systems, high data rates make high availability expensive to achieve. We observe, however, that unlike traditional data-processing applications that require precise recovery for correctness, many stream-processing applications can tolerate and benefit from weaker notions of recovery. Based on the correctness requirements of our target applications, we characterize various recovery types that differ in the level of recovery guarantees they offer. We describe how to adapt standard recovery techniques to provide these various guarantees. We also introduce a novel technique that leverages the data-flow aspects of streaming systems to achieve high availability at a low cost. Using analysis and simulations, we quantify the cost of offering each type of recovery guarantee and examine the performance of each recovery technique.

1 Introduction

Stream-processing engines [1, 3, 5, 6, 14, 16] are designed to support 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 and infrastructure monitoring (car traffic, air quality, computer networks), financial applications (stock-price monitoring, ticker failure detection), and asset tracking. In these applications, data sources are commonly located at remote sites or provided by different parties. Applications can thus achieve significant gains in scalability and efficiency when multiple distributed servers collectively process and aggregate data streams while routing them from their origins to the target applications. As a result, recent attention has been focused on extending stream processing to distributed environments [6, 7, 20].

A highly available service that seamlessly and quickly recovers from failure is a key goal of virtually all existing data management systems [8, 19, 22]. To achieve high availability in a distributed stream-processing system (DSPS), one could directly apply standard process-pair approaches [4, 9]. Traditional approaches, however, are impractical for most stream applications as high data rates cause the techniques to incur significant bandwidth and processing overhead.

Additionally, stream processing applications differ from traditional applications in their availability requirements: they often have weaker notions of correctness and can benefit from more lightweight high-availability schemes. Monitoring applications, for example, operate on data streams whose values are periodically refreshed and can tolerate tuple losses when a failure occurs. Applications that trigger alerts when certain combinations of events happen ("patient heart rate has changed to zero"), require that no tuples be lost, but may tolerate duplication. Of course, there are several stream-processing applications that require strong, precise recovery guarantees. Many financial applications require OLTP-style recovery guarantees and fall into this category.

In this paper, we thus first define recovery semantics for stream processing systems and derive a set of increasingly expensive recovery guarantees. Because properties

*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.

of operators, such as their possible non-determinism, affect the difficulty of achieving specific guarantees, we also develop a taxonomy for stream processing operators and for query-networks composed of these operators, based on their impacts on recovery semantics. Second, we present four failure-recovery algorithms that achieve some or all possible guarantees for an increasing runtime overhead. Third, using analysis and simulations, we comparatively study all four approaches. We analyze the cost of providing each recovery type for each query-network type and determine the best suited approach for that type of availability.

More precisely, when defining recovery guarantees, we distinguish and study three important types of failure recovery. (1) *Gap recovery* may lose tuples and state when a failure occurs. (2) *Rollback recovery* re-starts query-processing from a checkpoint. It does not lose tuples, but may produce redundant tuples. (3) *Precise recovery* takes over exactly from the point of failure, without any losses or duplication.

Commercial DBMSs typically offer precise or gap recovery capabilities [8, 19, 17, 18]. To the best of our knowledge, no existing solution addresses rollback recovery or a similar weak recovery model. Additionally, we argue that existing synchronous schemes for precise recovery are neither necessary nor preferable for stream-based applications due to their relatively high run-time overhead. Instead, we propose *asynchronous* solutions tailored to stream processing and capable of providing precise recovery guarantees at a lower overhead.

We present four failure-recovery techniques that differ in their runtime and recovery overheads. *Amnesia* is a lightweight scheme that provides gap recovery without any runtime overhead. *Passive standby* and *active standby* are adaptations of the standard process-pairs [4, 9] approach to stream processing. In passive standby, each primary server (a.k.a. node) asynchronously reflects its state update to its secondary node, which takes over from the latest checkpoint when the primary fails. In active standby, the secondary nodes process all tuples in parallel with their primaries.

As we demonstrate, the standby approaches that we propose still incur significant bandwidth and processing overhead and may become impractical when data rates are high. We therefore propose a new approach, *upstream backup*, that exploits the distributed and data-flow nature of stream processing to reduce the runtime operation overhead while trading off a small fraction of the recovery speed (and occasionally recovery guarantees). In this approach, upstream nodes 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. The main challenge in this approach lies in determining, at any given time, the maximum set of tuples logged upstream that can be safely discarded given the tuples produced on the node's output streams. This computation is particularly difficult given the many-to-many relationship between input and output tuples, the non-determinism of some operators, and the splits and merges of streams in the query network.

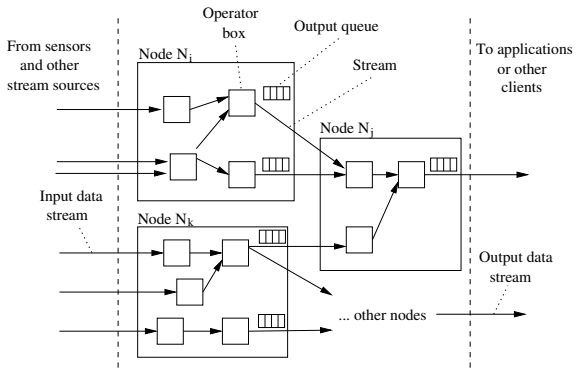


Figure 1. Query execution in a DSPS

All proposed approaches provide rollback recovery in their simple form and can be extended to provide precise recovery at a higher runtime cost.

The rest of this paper is organized as follows. We first provide background on stream processing in Section 2. In Section 3, we characterize three recovery types and develop a taxonomy of stream-oriented operators and queries based on their effects on recovery. In Sections 4 through 6, we devise four recovery techniques and describe how they support various recovery types for different types of operators and queries. In Section 7, we quantify the runtime overhead and recovery performance of the four high-availability approaches. We find that (1) for most queries, upstream backup provides precise recovery at a fraction of the runtime cost of others while keeping recovery time relatively short, (2) the size of query state and the frequency of tasks for high availability significantly influence recovery performance of upstream backup and runtime performance of passive standby, (3) recovery time *vs* runtime overhead is a clear tradeoff, and (4) each approach covers a complementary fraction of the solution space. We review related work in Section 8 and conclude in Section 9.

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, 3, 6] aim to address this "process-before-store" model and are building stream-processing systems.

In stream-processing applications, data streams are composed and aggregated by *operators* to produce output of interest. Operators are functions (map, filter, join, aggregate, etc.) that transform one or more input streams into one or more output streams. A loop-free, directed graph of operators such as shown in Figure 1 is called a *query network*.

Most stream-processing systems provide a standard set of stream-based operators [1, 3, 6] that users or application writers compose to create query networks.

A distributed stream-processing system (DSPS) partitions queries across multiple computing nodes and even multiple administrative domains. Each computing node locally runs a stream-processing engine. Figure 1 shows an example of a query distribution in a DSPS. In this figure, multiple streams arrive from various stream sources. The operators at nodes N_i and N_k first process these streams and transmit the resulting streams to node N_j and other nodes. Each node can have multiple output streams and can forward each stream 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.

A node associates each one of its output streams (streams crossing node boundary) with an *output queue* that can temporarily hold output tuples. Each high-availability approach uses these output queues differently, as discussed further in Section 5. Operators also have input queues through which they receive tuples. These queues are considered to be part of each operator's state and are not shown in the figure.

3 High-Availability Semantics

At any time, the state of a query network running at a node can be defined by a *snapshot*, which is a triple (I_k, E_k, O_k), where k indicates the number of state transitions that the query network went through since the beginning of the execution. I_k is a snapshot of the state of the query network's input streams. For each input stream S, $I_k[S]$ contains all tuples entered stream S but not yet consumed by the query network. E_k is a snapshot of the internal state of the query network, which consists of the states of operators and their input queues. O_k contains the ordered set of all tuples produced during the first k state transitions. A state transition occurs when an upstream node produces a tuple on an input stream or when an operator executes, changing its internal state, the state of its input queues, or outputting a tuple. Failure recovery also triggers state transitions. A sequence of snapshots forms an *execution history*.

3.1 Recovery Types

In this section, we propose three recovery types that differ in the execution histories they produce. The first, *gap recovery*, may lose input tuples during failure detection. Assuming that a failure happens at state $r-1$, the state r after failure satisfies: $O_r = O_{r-1}$ (the output does not change and no tuples are produced at state r), $E_r = \emptyset$ (the state of the query network is reset to the empty state), and $I_r = \emptyset$ (inputs are not saved, processing restarts on newly arriving tuples).

The second recovery type, *rollback recovery*, resets a query network to an earlier state (I_r, E_r, O_r), where $O_r = O_{r-1}$ (the output does not change). The internal state of the query network is reverted to an empty state ($E_r = \emptyset$) or an earlier state ($\exists k \in [0 : r-1] \ : \ E_r = E_k$)[1]. To avoid losing tuples, if the query network state is reverted, inputs must be reverted to an old-enough snapshot to contain all tuples in the lost state: I_{r-1} is a suffix of I_r and $\forall \text{tuple t} \in E_{r-1} \ : \ t \notin E_r \Rightarrow t \in I_r$.

	Before failure			After failure			
State	...	r-2	r-1	r+x	r+x+1	r+x+2	r+x+k
Recovery type							
-**Precise**	t_1	t_2	t_3	t_4	t_5	t_6	t_{k+4}
-**Gap**	t_1	t_2	t_3		t_5	t_6	t_{k+4}
-Rollback							
-**Repeating**	t_1	t_2	t_3	$\mathbf{t_2}$	$\mathbf{t_3}$	t_4	t_{k+2}
-**Convergent**	t_1	t_2	t_3	$\mathbf{t'_2}$	$\mathbf{t'_3}$	t_4	t_{k+2}
-**Divergent**	t_1	t_2	t_3	$\mathbf{t'_2}$	$\mathbf{t'_3}$	$\mathbf{t'_4}$	$\mathbf{t'_{k+2}}$

Figure 2. Outputs produced by each type of recovery

The third recovery type, *precise recovery*, produces an execution identical to an execution without failure: if (I_n, E_n, O_n) is the last snapshot of an execution with failure, then there exists at least one execution over the same input and without failure that has the same last snapshot. Precise recovery may be achieved by ensuring that the first state after failure is equal to the last state before failure or through rollback and elimination of redundant tuples.

The semantics of precise and gap recovery are independent of the operators in the query network. Rollback recovery may, however, offer different guarantees. We thus distinguish the following types of rollback recovery:

1. *Repeating recovery*: after a failure, a sequence of duplicate tuples appears in the output. Duplicate tuples are identical to original tuples produced before failure.
2. *Convergent recovery*: after a failure, a sequence of *redundant* tuples appears in the output, but the output eventually converges to the output without failure. A tuple is redundant if it is generated by re-processing the original input differently (e.g., by processing the original window with missing tuples, or by processing a new window due to different window alignments).
3. *Divergent recovery*: after a failure, a sequence of redundant tuples appears in the output. The output may not converge back to any output without failure, but it is still meaningful because it corresponds to an output produced by a query network initialized at the rolled-back position.

Figure 2 shows examples of outputs produced by each recovery type. Each t_i denotes a tuple, t'_i denotes a redundant tuple corresponding to t_i, and x state transitions are assumed between failure and recovery. k is any integer greater than 2.

[1] When the query network is non-deterministic, E_k may be an equivalent but possibly different earlier state

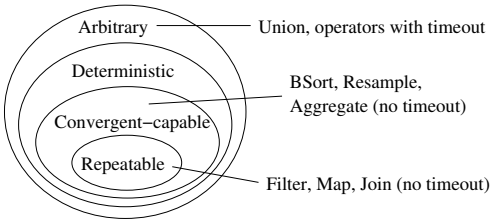


Figure 3. Taxonomy of Aurora operators

3.2 Operator Classification

We distinguish four types of operators based on their effects on recovery semantics: *arbitrary* (including non-deterministic), *deterministic*, *convergent-capable*, and *repeatable*. Figure 3 depicts the containment relationship among these operator types and the corresponding Aurora operators [1, 2] for each type. The type of a query network is defined to be the same as the type of its least general operator.

An operator is *deterministic* if it produces the same ordered output every time it starts from the same initial state and receives the same sequence of tuples on each input stream. There are three possible causes of non-determinism in operators: dependence on time (either execution time or input tuple arrival times), dependence on the arrival order of tuples on different input streams (e.g., Union), and use of non-determinism in processing such as randomization.

A deterministic operator is *convergent-capable* if it yields a convergent-recovery when restarted from an earlier snapshot of its inputs and an empty internal state. To be convergent-capable, an operator must thus rebuild its internal state from scratch and update it on subsequent inputs in a manner that eventually converges to the sequence of snapshots that would have existed without failure. Window alignment is the only possible cause that prevents an operator from being convergent-capable. This is because window boundaries define the sequences of tuples over which operators perform computations and update their states. Therefore, a deterministic operator is convergent-capable if and only if its window alignments always converge to the same alignment when restarted from an arbitrary one.

A convergent-capable operator is *repeatable* if restarting the operator from a previous snapshot of the inputs and an empty state results in a repeating recovery (the operator must produce duplicate rather than redundant tuples). A necessary condition for an operator to be repeatable is for the operator to use at most one tuple from each input stream to produce an output tuple. If a sequence of multiple tuples contributes to an output tuple, then restarting the operator from the middle of that sequence may yield at least one different output tuple. Aggregates are therefore not repeatable in general while Map and Filter are repeatable because they have one input stream and they process each input tuple independent of others. Join (without timeout) is also repeatable because the windows defined on input streams have alignments relative to the latest input tuple being processed.

$\mathbf{\Delta}$	Number of lost or redundant tuples
$\mathbf{D}$	Failure detection delay
$\mathbf{K}$	Delay before processing first duplicate input tuple
$\mathbf{Q}$	Average number of input tuples to re-process
$\mathbf{R}$	Time to reconstruct the query network
$\mathbf{r}$	Time to redirect input streams
$\mathbf{d}$	Network transmission delay between any two nodes
$\mathbf{M}$	Checkpoint or queue-trimming interval
λ	Input tuple arrival rate
$\mathbf{C}$	Size of checkpoint message
$\mathbf{c}$	Size of queue-trimming message
$\mathbf{nb_ops}$	Number of operators in the query network
$\mathbf{nb_paths}$	Number of paths from input to output streams

Table 1. Summary of notation

3.3 Recovery Time

We assume that each node has a statically assigned recovery node and that it takes a constant time D for a recovery node to detect the failure of its pre-assigned primary (for instance, the recovery node can periodically send keep-alive messages to its primary). We define *recovery time* as the interval between the time the recovery node learns about the failure and the time it reaches the pre-failure state of the primary (or an equivalent state for non-deterministic networks). Recovery time thus measures the time spent recreating the failed state.

In the following three sections, we present high-availability approaches that provide single-fault tolerance with gap recovery, rollback recovery, and precise recovery, respectively. For each approach, we discuss the impact of the query-network type on recovery and analyze the trade-offs between recovery time and runtime overhead.

4 Gap Recovery

The simplest approach to high availability involves restarting failed query networks from an empty state and from the latest tuples produced by upstream nodes. This approach, called *amnesia*, produces a gap recovery for all types of query networks. Amnesia is well suited for applications that tolerate tuple loss and require a lightweight approach to continue processing in spite of node failures. The failure detection delay, the rates of tuples on streams, and the size of the state of the query network determine the number, Δ, of lost tuples. This approach has no extra runtime overhead for high availability.

For amnesia, because the failed state is not recreated and input tuples are dropped while recovery node is not ready to accept them, recovery time is zero. It does take time R to reconstruct the failed query network and time r to redirect the inputs to the recovery node, but when processing restarts, the first tuples processed are those that would have

	Query-network type			
Approach	**Repeatable**	**Convergent-capable**	**Deterministic**	**Arbitrary**
Passive standby	Repeating	Repeating	Repeating	Repeating
Upstream backup	Repeating	Convergent	Divergent	Divergent
Active standby	Repeating	Repeating	Repeating	Divergent

Table 2. Type of rollback recovery achieved by each high-availability approach for each query-network type

	recovery time	**bw_overhead**	**proc_overhead**
Amnesia	0	0	0
Passive standby	$K = r + d \ ; \ Q = \frac{M\lambda}{2}$	$f_1(\frac{1}{M}, C)$	$f_2(\frac{1}{M}, C)$
Upstream backup	$K = R + r + d \, ; Q = \text{tuples}(\text{E}_\text{i}) + M\lambda + 2d\lambda$	$f_3(\frac{1}{M}, c)$	$f_4(\frac{1}{M}, \text{nb_ops}, \text{nb_paths})$
Active standby	ϵ (negligible)	$100\% + f_3(\frac{1}{M}, c)$	$100\% + 2 * f_4(\frac{1}{M}, \text{nb_ops}, \text{nb_paths})$

Table 3. Recovery time and runtime overhead for each approach

been processed at the same time if the failure did not happen. There is therefore no extra delay caused by failure and recovery.

For all the approaches that we investigate, if primary nodes have statically assigned recovery nodes, the latter can cache an inactive snapshot of the fully constructed query networks. By doing so, when the failover node takes over, it is already running a stream-processing engine (no new processes need to be started) and it already holds the fully constructed query network, reducing R to zero.

5 Rollback Recovery Protocols

In this section, we describe three approaches to achieve different types of rollback recovery. We first present *passive standby*, an adaptation of the process-pairs model with passive backup. As a second approach, we introduce a new algorithm called *upstream backup*, where upstream nodes in the processing flow serve as backup for their downstream neighbors. We finally describe *active standby*, the second adaptation of the process-pairs model where the standby performs all processing in parallel with the primary. We defer the discussion of active standby to the end, because it relies on some concepts introduced in upstream backup.

For each approach, we examine (1) the recovery types it provides, (2) the average recovery time, and (3) the runtime overhead for high availability. We divide the runtime overhead into processing and communication (or bandwidth) overhead. Table 1 summarizes our notation. Table 2 summarizes the recovery types achieved by each approach. Table 3 summarizes the recovery time and overhead metrics.

5.1 Passive Standby

Passive Standby associates each primary node with a standby node and handles failures at the granularity of nodes. A primary periodically sends an update of its latest snapshot (I_k, E_k, O_k) to the secondary. Until the secondary acknowledges after reflecting the state update to itself, the primary does not forward output tuples to downstream neighbors while it keeps processing. Passive Standby is asynchronous in that checkpointing doesn't block ordinary processing. The frequency, M, of sending state updates is thus configurable to any value. Sent to downstream neighbors, tuples still remain in output queues until they are acknowledged by downstream nodes. The latter acknowledge only *after* processing the tuples and checkpointing the results to their secondary nodes.

When a primary fails, the secondary takes over and sends all tuples from its output queues to the primary's downstream neighbors. At the same time, the failed primary's upstream neighbors redirect their outputs (including tuples in their output queues) to the secondary, which starts processing these input streams starting from the latest tuples acknowledged by the previous primary. When the old primary comes back up, it becomes the new secondary.

Recovery Type: Passive standby produces a repeating recovery for all query networks because the primary always copies its state to the secondary before sending any new tuples to downstream nodes. When the secondary takes over, it restarts exactly from a past state of the failed primary.

Recovery Time: The main advantage of passive standby is a short failure recovery time, achieved by having the backup hold a complete and recent snapshot of the primary's state. When the secondary takes over, no new process needs to be started. The secondary already runs a stream processing engine with a complete query network, and simply a paused operator scheduler. Since recovery consists of re-processing tuples, recovery time is equal to $K + Qs$, where K is the delay before the recovery node receives its first input tuple, Q is the number of duplicate input tuples that must be reprocessed, and s is the average processing time per tuple. For passive-standby, $K = r + d$. r is the time to redirect input streams and d the time for the first tuple to propagate from the upstream nodes. Q is on average half a checkpoint interval worth of input tuples. s depends on the complexity of the query network. Because the primary sends tuples in a batch, after every checkpoint, the average number, Δ, of duplicate tuples is close to $M\lambda_{out}$, where λ_{out} is the rate of tuples on output streams.

Overhead: The main drawback of passive standby is its high overhead. The bandwidth overhead is inversely proportional to the checkpoint interval and proportional to the size of checkpoint messages. The processing overhead consists of generating and processing checkpoint messages (proportional to the bandwidth overhead). The checkpoint interval, M, therefore determines the tradeoff between runtime overhead and recovery time. Table 3 summarizes the recovery time and overhead of passive standby.

5.2 Upstream Backup

Given the overhead of process-pair-based approaches, we propose an alternate model that leverages the network-flow form of stream 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 in Figure 1, 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 re-processing the logged tuples.

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. Tuples are trimmed (i.e., removed) once they have been fully processed by all downstream neighbors *and* tuples resulting from that processing have been transferred further down the network. However, queue trimming is nontrivial in general due to many-to-may relationship between input and output tuples, out-of-orderness of streams, and non-deterministic operators.

Figure 4 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. That neighbor, N_j for instance, can then notify its own upstream neighbor (N_i) that the tuples that contributed to producing the acknowledged tuples can now be trimmed. These notifications are called level-1 acknowledgments (denoted $\mathrm{ACK}(1, S_i, u)$, where S_i identifies a stream and u identifies a tuple on that stream), since using them to trim output queues allows the system to tolerate any *single* failure. Leaf nodes in the query graph use level-0 acknowledgments to trim their output queues.

5.2.1 Queue Trimming Protocol

We first describe the details of queue-trimming message (level-1 acknowledgments) generation. Assume the existence of a function that determines, for a given node, which

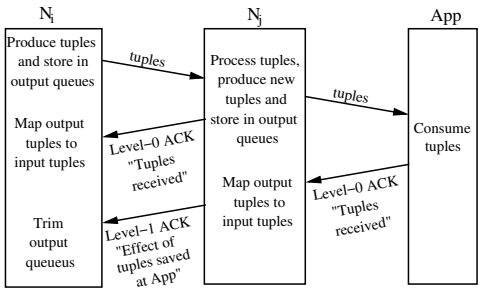


Figure 4. Overview of inter-node communication in upstream backup

input tuples caused the generation of some output tuples. We define this function as $\mathrm{cause}((S_o, v), S_i) \rightarrow (S_i, u)$, where S_i and S_o identify streams, and u and v identify tuples on these streams. Given an output tuple identifier (S_o, v) (we use $S_o[v]$ to denote the actual tuple) 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 5.2.2.

Each node generates a queue-trimming message only after its downstream neighbors return level-0 acknowledgments that confirm their tuple reception. Nodes produce level-0 acknowledgments for every M_n tuples they receive, every M_s seconds, or whichever occurs first. A lower acknowledgment frequency reduces bandwidth utilization, but increases the size of output queues and the recovery time.

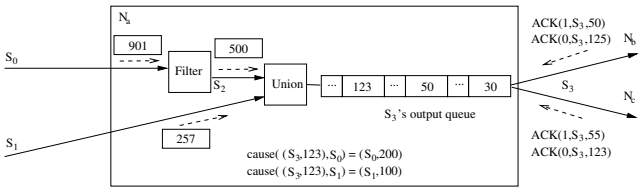
When *all* downstream neighbors of a node produce level-0 acknowledgments for a tuple v (or following) on a stream S_o, the node maps (S_o, v) back onto the earliest input tuple, on each input stream S_i, that caused it. I.e., $\mathrm{cause}((S_o, v), S_i)$. For each input stream, the node identifies the most recent tuple whose effect has propagated in such a manner *on all output streams* and produces a level-1 acknowledgment for that tuple. To avoid spurious transmissions, level-1 acknowledgments can be produced every M_n level-0 acknowledgments or every M_s seconds. Figure 5 shows the pseudocode for generating level-1 acknowledgments. In the figure, N denotes the set of downstream neighbors, and S_{input} and S_{output} the sets of input and output streams. The $\mathrm{down_bound}$ data structure holds, for each output stream, the latest tuple acknowledged at level-0 by each downstream neighbor. $\mathrm{min_downbound}$ holds the latest tuple acknowledged by all these neighbors. Similarly, $\mathrm{up_bound}$ and $\mathrm{min_upbound}$ are used to keep track of input tuples for which level-1 acknowledgements can be issued.

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.

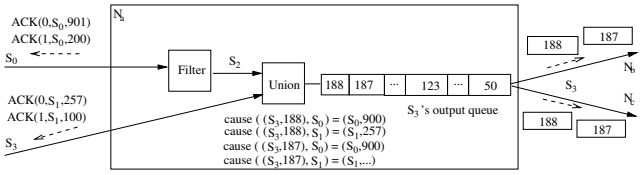
Figure 6 illustrates one iteration of the upstream-backup algorithms on one node. Figure 6(a) shows node N_a

```
01. repeat forever
02.     wait to receive an ACK(0, S_o, v) from N_i
03.         down_bound[S_o, N_i] ← v
04.         w ← min{down_bound[S_o, N_i] : N_i ∈ N}
05.         if w > min_downbound[S_o]
06.             min_downbound[S_o] ← w
07.             ∀S_i ∈ S_input such that ∃ a path from S_i to S_o
08.                 up_bound[S_i, S_o] ← cause((S_o, w), S_i)
09.                 x ← min{up_bound[S_i, S_o] : S_o ∈ S_output}
10.                 if x > min_upbound[S_i]
11.                     min_upbound[S_i] ← x
12.                     acknowledge ACK(1, S_i, x)
```

Figure 5. Algorithm for generating level-1 acknowledgments



(a) A node receives acks from its downstream neighbors, while receiving new tuples from its upstream neighbors. The filter operator also produces $S_2[500]$ which corresponds to $S_0[900]$.



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

Figure 6. One iteration of upstream-backup

receiving level-0 and level-1 acknowledgments from two downstream neighbors N_b and N_c: $\mathrm{ACK}(0, S_3, 125)$, $\mathrm{ACK}(0, S_3, 123)$, $\mathrm{ACK}(1, S_3, 50)$, and $\mathrm{ACK}(1, S_3, 55)$. 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 acknowledge these tuples at level-0, assuming $M_n = 1$.

5.2.2 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 $(S_i, u) \leftarrow \mathrm{cause}((S_o, v), S_i)$. 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 applies but operators must be able to identify, at any given time, the first tuple on each of their input streams required to regenerate their complete current state (usually it is the tuple marking the beginning of the current window). 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-offset indicators* to tuples as they travel through operators on a node. For a tuple $S[w]$ (on an intermediate stream or an output stream), these indicators, denoted with $\mathrm{indicators}(S, w)$, contain the identifiers of the oldest tuples on *input streams* necessary to generate $S[w]$. When an operator produces a tuple, it determines the oldest tuple for each of its input queues that has contributed to the state yielding the new tuple. The operator reads the indicators from each such input tuple and appends them to the indicators of the output tuple. 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 would only append the indicator for the stream from which the output tuple was issued. *Nodes do not forward indicators to their downstream neighbors*. When a tuple $S_i[u]$ enters a node, its indicators are initialized to the tuple itself. Figure 7 presents the algorithm for managing indicators.

Applying the cause function thus consists in reading the set of indicators of a tuple in the output queue: i.e., $\mathrm{cause}((S_o, v), S_i) = \{(S_x, y) \in \mathrm{indicators}(S_o, v) : x = i\}$. Due to operators such as union however $\mathrm{cause}((S_o, v), S_i))$ may return an empty set even though a path from S_i to S_o exists. In that case, the value used should be the first non-empty set returned by the cause function for the tuples preceding v on S_o. If the cause function returns multiple indicators for one stream (because the stream contributes to the output tuple through multiple paths), the oldest is used.

Operators with low selectivity may introduce *flow tuples* (i.e., null output tuples) which are forwarded unchanged through all the operators to reduce the granularity of queue trimming. Flow tuples also do not cross node boundary.

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, $\mathrm{indicators}(S_2, 500) = \{(S_0, 900)\}$. In Figure 6(b), the

```
1. // At each operator b
2. for each output tuple produced b.S_o[v]
3.   ∀b.S_i ∈ b.S_input (contributing to b.S_o[v])
4.     get oldest state-contributing tuple (b.S_i, u)
5.     if indicators(b.S_i, u) = ∅ // if tuple S_i[u] has crossed node boundary
6.       indicators(b.S_i, u) ← {(b.S_i, u)}
7.     indicators(b.S_o, v) ← indicators(b.S_o, v) ∪ indicators(b.S_i, u)
```

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

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, $\text{indicators}(S_3, 187) = \{(S_0, 900)\}$ and $\text{indicators}(S_3, 188) = \{(S_1, 257)\}$. Therefore, $\text{cause}((S_3, 188), S_0) = (S_0, 900)$, $\text{cause}((S_3, 188), S_1) = (S_1, 257)$, and $\text{cause}((S_3, 187), S_0) = (S_0, 900)$. $\text{cause}((S_3, 187), S_1)$ depends on the indicators on the tuples preceding S_3.

Recovery Type: Upstream backup restarts from an empty state producing a repeating recovery for repeatable query networks, a convergent recovery for convergent-capable query networks and a divergent recovery for all others. These guarantees are weaker than those of passive standby.

Recovery Time: The lost state is recreated by re-processing input tuples. The time, K, to receive the first tuple can be longer than for passive standby because of the time, R, required to reconstruct the failed query network. This delay can be avoided by having the recovery node cache an inactive snapshot of the primary's query network. In upstream-backup, the recovery node may re-process significantly more tuples than passive standby to recreate the lost state. It must re-process (1) all tuples that contributed to state E_i (i.e., $\text{tuples}(E_i) = \{t | t \in E_i\}$), (2) a complete queue-trimming interval worth of tuples on average (due to the periodic transmission of both level-0 and level-1 acknowledgments), and (3) some extra tuples that account for the propagation delays added by level-0 acknowledgments. Table 3 summarizes these recovery time parameters for upstream backup. Finally, the number, Δ, of redundant tuples is the product of the number of tuples to reprocess (Q) and the query-network selectivity minus the number of tuples that remain as part of the query-network state.

Overhead: Upstream backup has the lowest runtime bandwidth overhead because queue-trimming messages are significantly smaller than checkpoint messages of the other approaches. These queue-trimming messages contain only the tuple identifiers for streams crossing node boundaries. The processing overhead is also small: operators keeps track of the oldest tuple (and its indicators) on each of their input streams that contributes to their current states. Furthermore, we can reduce the spatial and computational overhead of managing indicators by processing them and appending them to tuples occasionally. In general, the total overhead, as summarized in Table 3, is proportional to the number of operators and the number of paths, where a path is a data flow connecting an input stream to an output stream.

5.3 Active Standby

The third approach, *active standby*, is a variation of process pairs but uses some features of upstream backup. In active 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. In general, this parallel processing guarantees faster recovery but weaker recovery semantics than passive standby.

A secondary node logs the tuples it produces in output queues without sending them downstream. If the primary fails, the secondary takes over and resends the logged tuples to all downstream neighbors. For non-deterministic query networks, the primary and secondary may have different tuples in their output queues, complicating the task of bounding the queues at the secondary. We tackle this problem by exploiting queue-offset indicators introduced in Section 5.2.2. When a primary receives a level-0 acknowledgment 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*. Based on that message, the secondary determines the trimming bound for each output queue.

We use Figure 6 to illustrate active standby. When $\text{ACK}(0, S_3, 125)$ and $\text{ACK}(0, S_3, 123)$ 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 S_3) for which $\text{cause}((S_3, u), S_0)$ returns a tuple older than $(S_0, 200)$ and $\text{cause}((S_3, u), S_1)$ returns a tuple older than $(S_1, 100)$.

Recovery Type: Because the secondary processes tuples in parallel with the primary and takes over from the latest checkpoint, active standby provides repeating recovery for all deterministic query networks and divergent recovery for others. These guarantees are thus better than those of upstream backup, but worse than those of passive standby.

Recovery Time: Because the standby keeps processing during failure, it only needs to transmit all redundant tuples in its output queue to reach a state equivalent to that of the primary. Recovery time is therefore negligible. The number, Δ, of redundant tuples is on average $\frac{M\lambda_{out}}{2} + 2d\lambda_{out}$ for each output stream, since M determines the trimming interval for the secondary's output queues.

	Passive standby		
Q. network	Bandwidth	Processing	Rec. time
Arbitrary	none	negligible	none
	Active standby		
Q. network	Bandwidth	Processing	Rec. time
Deterministic	none	negligible	r
Arbitrary	freq. small checkpoints	freq. small checkpoints	r
	Upstream backup		
Q. network	Bandwidth	Processing	Rec. time
Repeatable	$\frac{f(k)*\text{size(tuple_id)}}{\text{size(tuple)}}$	negligible	none
Convergent	$\frac{f(k)*\text{size(tuple_id)}}{\text{size(tuple)}}$	double	negligible
Arbitrary	N/A	N/A	N/A

Table 4. Additional runtime overhead and recovery time for precise recovery

Parameter	Meaning	Default
Stream Rate	# of tuples generated by a stream source per sec (tuples/sec)	1000
Tuple Size	size of a tuple and tuple id (bytes)	50, 8
Network	bandwidth(Mbps) and delay(msec)	16, 5
Proc. cost	avg. processing time per input tuple (μsec)	
Aggregate	$10 * \text{Window} * \frac{1}{\text{Advance}}$	100
Filter	—	10
Selectivity	expected value of $\frac{\text{\# of output tuples emitted}}{\text{\# of input tuples consumed}}$	0.1
F. D. Delay	delay to detect the failure of a node (msec)	100
Comm. Int.	queue-trimming/checkpoint interval (msec)	50

Table 5. Simulation parameters

Overhead: Because all processing is replicated by the standby node, both proc_overhead and bw_overhead are approximately 100%. These overheads are actually even a little higher due to processing offset indicators and transmitting queue-trimming messages. Table 3 summarizes the results for active standby.

6 Precise-Recovery Extensions

For convergent-capable query networks, each approach described above can achieve precise recovery by eliminating redundant tuples during convergence. It is also possible, though much more costly, to provide precise recovery for arbitrary networks. Table 4 summarizes the extra runtime overhead and recovery time required for precise recovery.

Passive standby provides repeating recovery for all types of query networks. Redundant tuples are therefore identical duplicates. To make recovery precise, before sending any output tuples, the failover node must ask downstream neighbors for the identifiers of the last tuples they received and discard all tuples preceding the ones identified. These requests can be made while the recovery node regenerates the failed state, achieving precise recovery without added overhead.

For active standby and deterministic query networks, a similar procedure makes recovery precise. The delay necessary to ask downstream nodes for the latest tuples received can not be masked though and extends recovery by one round-trip between nodes (r). For other query networks, to achieve precise recovery, we must ensure that both the primary and secondary process tuples in an identical fashion. To do so, whenever a non-deterministic operator is scheduled, the primary must communicate to the secondary how the operator's input, state, and output change. These messages are equivalent to small but frequent checkpoints.

Providing precise recovery is most costly for upstream backup and increasingly so with the complexity of the query-network type. In repeatable query networks, operators produce output tuples by combining at most one tuple from each input stream. Offset-indicators therefore uniquely identify output tuples and can be used for duplicate elimination. Each node thus must forward these indicators along with output tuples. The extra overhead is approximately $\frac{f(k)*\text{size(tuple_id)}}{\text{size(tuple)}}$, where $f(k)$ is a function of the average number of input streams (at a node) that contribute to an output stream. $f(k)$ could be k simply, but optimizations are possible.

For convergent query networks, a recovery node must determine when the output has converged to the output without failure. One approach is to use additional offset indicators to mark the most recent tuples that contribute to the query-network state. A recovery node then discards all output tuples until the most recent output tuples seen before failure are output again. This approach doubles the number of offset indicators to process, thus doubling the processing overhead. The added bandwidth overhead necessary for precise recovery is the same as with repeatable query networks.

Upstream backup is not well suited to provide precise recovery for more complex query networks. It would be possible to do so by checkpointing non-deterministic operators' states and results along with queue-trimming messages, but the overhead of this would be too high.

7 Evaluation

In this section, we evaluate the performance of each approach through simulations. Using CSIM [15], we built a detailed simulator that models a DSPS with stream sources, stream sinks, and processing nodes that run (parts of) query networks. Tables 5 summarizes the main simulation parameters. Each point shown in the figures is an average of 25 independent simulations, at least one simulated minute each.

Amnesia has no overhead and a zero recovery time but provides only gap recovery. We therefore examine the performance of the other three approaches for rollback recovery and evaluate the added overhead of achieving precise recovery. We also examine the effect of the query-network type and other query-network properties on runtime overhead and recovery time. For the overhead measurements, we only present bandwidth overhead because processing overhead poses similar tradeoffs, while being more difficult to measure accurately. We refer the reader to Sections 4 through 6 for a detailed discussion of relative processing overheads.

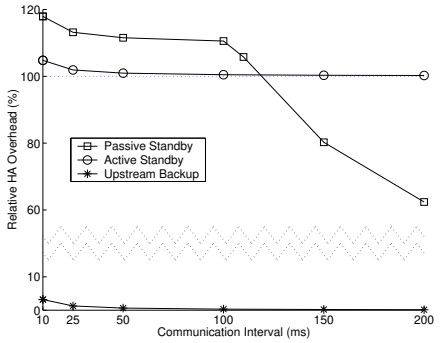


Figure 8. Runtime overhead of each approach for increasing communication intervals

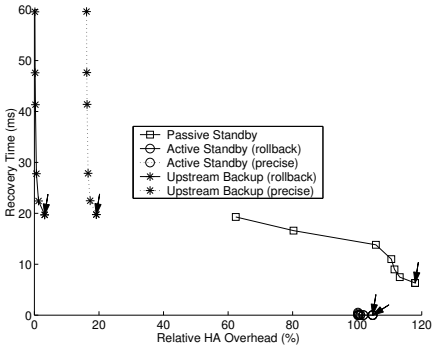


Figure 9. Recovery time and runtime overhead for rollback and precise recovery as the communication interval ranges from 10msec (indicated with arrows) to 200msec

7.1 Runtime Overhead vs Recovery Time

We first examine the runtime overhead and recovery time tradeoffs for each approach when providing rollback recovery. Because most stream-processing operators are convergent-capable, we use an aggregate operator as the basis for comparison. We simulate a single aggregate operator with the default parameter values specified in Table 5 (this aggregate consumes 10% of a node's processing capacity).

The only tunable parameter for each approach is the communication interval, which is basically the queue-trimming interval for upstream backup and active standby and the checkpointing interval for passive standby. For all approaches, the overhead decreases as the communication interval increases, as shown in Figure 8. Upstream backup has bandwidth overhead close to zero even for a small communication interval. With a 10msec interval, the node transmits one 8-byte tuple identifier for every 10 tuples it receives, yielding an overhead of 1.6%. Active standby has an overhead of at least 100% because the secondary receives all input tuples in parallel with the primary. The small overhead component above 100% is due to the queue-trimming messages used to discard output tuples from the secondary. Only passive standby's overhead varies significantly with the communication interval. In passive standby, each checkpoint message contains an update of the complete state of the network. When operators have a selectivity of less than 1.0, increasing the interval between checkpoints also increases the number of tuples processed and dropped without being checkpointed. The knee at 100msec corresponds to the 100msec window size. The curve would be smoother for a larger query network.

Looking at bandwidth overhead, upstream backup is a clear winner. This advantage comes at the cost of a slower recovery. Figure 9 shows the tradeoff between recovery time and bandwidth overhead for each approach as we vary the communication interval from 10msec through 200msec. Upstream backup has the longest recovery time since the recovery node recreates the complete state of the failed query network. The approach is also most sensitive to the length of the communication interval. Frequent trimming reduces recovery time for a negligible added overhead, but the size of the query network eventually limits the recovery speed. Passive standby recovers faster because the secondary already has a snapshot of the last checkpoint and must re-process only half a checkpoint worth of tuples on average. Active standby has a negligible recovery time. The secondary only needs to resend half a queue-trimming interval worth of duplicate tuples stored in its output queues.

In this simulation, we ignored r, the time to redirect input streams from the primary to the failover node. This delay adds a constant to the recovery times of both upstream backup and passive standby (about 40msec in our prototype implementation).

7.2 Cost of Precise Recovery

Figure 9 also presents the recovery time and runtime overhead when recovery is precise. For active and passive standby, precise recovery of convergent-capable query networks comes at a negligible incremental cost compared with rollback recovery. Precise recovery increases the overhead of upstream backup by a little over 16% (equal to: $\frac{k*\text{size(tuple_id)}}{\text{size(tuple)}}$, with $k = 1$, and $\frac{\text{size(tuple_id)}}{\text{size(tuple)}} = \frac{8}{50} = 0.16$). The overhead thus remains much lower than that of the process-pair based approaches.

For all approaches, the precise recovery time is almost the same as the rollback recovery time. Upstream backup must now process additional offset indicators but this adds negligible delay. Additionally, for all approaches, recovery nodes must now ask downstream neighbors for the latest tuples they received. For upstream backup and passive standby this communication proceeds in parallel with tuple re-processing (or input stream redirection). For active standby the delay cannot be masked and extends recovery by a constant value r (40msec in our prototype). In the simulations, however,

Query Network Type	Result	Upstream Backup	Active Standby	Passive Standby
Repeatable	Bw overhead (%)	0.64	100.96	101.27
	Rec. time (msec)	7.35	2.07	5.85
Convergent-capable	Bw overhead	0.64	100.95	111.55
	Rec. time	27.79	0.07	9.00
Non-deterministic	Bw overhead	1.27	101.91	101.90
	Rec. time	8.66	1.98	7.24

Table 6. Effects of query-network type

we ignore the time to redirect input and output streams and with $r = 0$, active standby has a zero recovery time because it drops duplicate tuples without retransmitting them.

7.3 Effects of Query-Network Type

The previous section examined the difference in performance between precise recovery and rollback recovery for convergent-capable networks. In this section, we examine the effects of query-network types on the basic performance of rollback recovery. Table 6 summarizes the recovery time and bandwidth overhead of each approach when the query network consists of a repeatable filter with selectivity 1.0, our default convergent-capable aggregate, and a non-deterministic union operator that takes two input streams (500 tuples/sec each) and merges them into one output stream. The results show that the overhead of passive standby and the recovery time of upstream backup are significantly higher for the aggregate (111% and 28msec, respectively) than for the other two operators (101% and 8msec approximately). Interestingly, the overhead and recovery values for union and filter are almost identical for all approaches.

Overheads of all approaches are independent from the query-network type. Upstream backup and active standby use queue-trimming messages. Their overheads thus depend on the relative rates of queue-trimming messages and tuples on input streams rather than any other property of the query network. In Table 6, the union has a higher overhead with these approaches because it has two input streams at half the rate each. The overhead of passive standby is proportional to the size of the checkpoint messages, which does not depend on the type of the query network but on the magnitude of changes in query-network state between two checkpoints.

Recovery times of all approaches are independent of the query-network type. Active standby recovers by retransmitting output tuples. For instance, in Table 6, the output rate is ten times lower for the aggregate because of the 10msec advance, resulting in a faster recovery. The other two approaches recover by re-processing tuples. Passive standby re-processes half a checkpoint worth of tuples on average. Its recovery performance is thus independent of the type of the query network but rather depends on processing complexity (during recovery tuples are re-processed at the maximum rate). Upstream backup's recovery also depends on processing complexity. There is, however, a second parameter. The number of tuples that upstream backup must reprocess depends on the size of the query-network state.

Hence, the type of the query network does not affect the recovery time or runtime overhead. Rather, the size of the query-network state and the rate and magnitude of the state changes affect (1) the recovery time of upstream backup and (2) the overhead and recovery time of passive standby.

Window size (tuples)	100	200	300	400	500
PS overhead (%)	111.6	111.6	111.6	111.6	111.6
PS rec. time (msec)	9.0	11.7	15.4	19.0	24.2
UB rec. time (msec)	27.8	56.6	94.6	139.7	195.9

Table 7. Effects of query-network state size

Advance (tuples)	100	50	25	10	5
PS overhead (%)	102.6	103.6	105.6	111.6	121.5
PS rec. time (msec)	7.7	7.5	7.7	9.0	12.3
UB rec. time (msec)	22.0	21.6	21.3	27.8	40.5

Table 8. Effects of rate of query-network state change

7.3.1 Size of Query-Network State

We examine the effects of increasing the size of the query-network state by simulating the failure and recovery of an aggregate operator with increasing window size (100 to 500 tuples), but a constant 10-tuple advance. Table 7 shows the resulting passive standby (PS) overhead and both passive standby and upstream backup (UB) recovery times.

Increasing the size of the query-network state does not necessarily increase the rate at which that state changes. In this experiment, the overhead of passive standby remains constant at 111%. The recovery time of passive standby increases by about a factor of three when the size of the state quintuples. This increase is due to the heavier per-tuple processing cost, due to computing aggregate values over larger numbers of tuples. The increase in recovery time is more pronounced for upstream backup. The recovery time increases roughly linearly with the size of the state. Upstream backup must indeed re-process a number of tuples directly proportional to the size of the query-network state.

7.3.2 Rate of Query-Network State Change

We examine the impact of varying the rate at which the state of a query network changes by simulating the failure and recovery of an aggregate operator with decreasing window advance from 100msec to 5msec. By doing so, we also increase the selectivity from 0.01 to 0.2. Table 8 shows the impact of this increase in selectivity and query-network state change rate on the overhead of passive standby and the recovery times of both passive standby and upstream backup.

As expected, the overhead of passive standby increases proportionally to the magnitude of changes in query-network state. The advance determines the number of tuples that the operator produces in the 100msec checkpoint interval. This number increases from 1 to 20 as the advance decreases from 100msec to 5msec.

The increased per-input-tuple processing cost due to a smaller advance, slightly prolongs recovery for passive standby (visible for an advance of 10 tuples of less). We might expect the same effect to cause a slight increase in the recovery time of upstream backup. We measure a decrease instead. Upstream backup periodically updates the identifiers of the oldest tuples on each input stream that contribute to the current query-network state. When the state changes more rapidly, the older tuple are discarded faster and recovery restarts from a later point. This in turn results in a faster recovery. For a small enough advance, however, the added processing cost dominates recovery time. As the advance reaches 10msec, the recovery time starts increasing.

In summary, for rollback recovery, the *type* of the query network has no impact on recovery time and runtime overhead. Furthermore, active-standby's overhead and recovery time are both independent of any query network characteristics. Upstream backup's communication overhead is also independent of the query network but the size of the query-network state increases upstream backup's recovery time. In contrast, the rate and magnitude at which that state changes impacts the runtime overhead of passive standby.

7.4 Effect of Network Size

Increasing the number of operators in a query network increases the size of the query-network state, the rate at which this state changes, and the complexity of processing any single tuple. Figure 10 shows the performance of each approach for a chain of 1 to 5 aggregate operators (with the default parameter values listed in Table 5). Other configurations yield similar results.

As expected, increasing the number of operators increases the overhead of passive standby because the number of tuples that are produced inside or at the output of the query network increases. Larger query networks also slightly increase recovery time for passive standby. The recovery time of upstream backup increases more rapidly, achieving a little over 120msec for 5 operators. Interestingly, even with a larger query network, upstream backup still provides precise recovery at a fraction of the cost of the other approaches.

7.5 Discussion

The experimental results reveal that each approach poses a clear tradeoff between recovery time and processing overhead. Active standby, with its high overhead and negligible recovery time, appears particularly well suited for systems

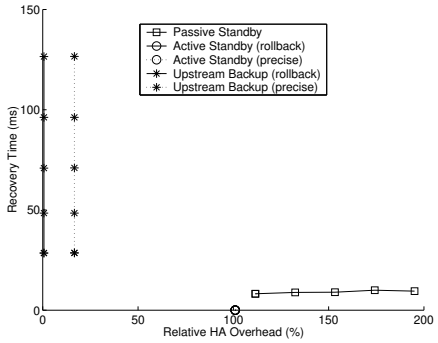


Figure 10. Effects of the number of operators

where failures are frequent or quick recovery justifies high runtime costs.

Passive standby does not seem well suited to stream-processing systems as its performance is worse than that of active standby for both recovery time and runtime overhead. Passive standby, however, is the only approach that easily provides precise recovery for arbitrary query networks.

Upstream backup provides precise recovery for most query networks with lowest runtime overhead but at the cost of a longer recovery. The recovery time of this approach however can be significantly reduced by distributing the recovery load over multiple nodes. In general, upstream backup is appropriate when server failures are infrequent and short recovery delays are tolerable. Contrary to process-pair approaches, recovery nodes can be chosen among live nodes allowing all servers to process data streams at runtime.

With all three approaches, the failure of the first server in a sequence may result in lost tuples (especially for upstream backup), unless data sources are capable of temporarily logging their output tuples. The problem can be overcome by logging all tuples entering the system on two machines with the second machine acting as passive or active standby for these input streams, or as dummy upstream backup.

8 Related Work

Recently, there has been much work on data-stream processing (e.g., Aurora [1, 5], STREAM [16], TelegraphCQ [6]), including proposals for distributed engines [7, 20]. The basic idea of DSPSs is to partition logical processing networks into multiple sub-networks and execute them on distributed machines to improve scalability, robustness, and performance.

The process-pairs model is the standard high availability approach adopted by virtually all existing DBMSs [8, 19]. One of most advanced such recovery facility is Oracle10*g*/DataGuard [17]. It is built on top of Oracle Streams [18], which enables cross-database event propagation and trigger-rule-based processing of event streams. DataGuard supports three data recovery modes, all of which work by transmitting redo records from the primary to the

standby servers: maximum protection (MPR), maximum availability (MAV), and max performance (MPE). MPR synchronously applies the same update to multiple machines as part of the same transaction, providing precise recovery. MPE asynchronously transmits transaction redo logs to the standby, providing gap recovery only. MAV is a hybrid scheme that switches between MPR and MPE depending on the accessibility of the standby. In contrast to the DataGuard recovery facilities, all of our approaches can provide precise recovery guarantees without significant degradation in performance because they are asynchronous and only require coarse-grained communication between servers.

Commercial workflow systems [11] also rely on redundant components to achieve high availability. A variation of the process-pairs approach is used in the Exotica workflow system [12]. 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 [13], 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.

In parallel processing systems, router nodes distribute incoming messages, potentially after processing them, across a set of parallel servers [10, 20]. If a server fails, the router re-directs incoming messages to other nodes. These approaches address how to select failover nodes and re-route messages to them, whereas we focus on replicating and recovering state. In MQSeries [10], messages that are being processed by a server when the failure happens are trapped until the server recovers. Flux [21] introduces a high-availability technique similar to our active-standby method. It tries to accomplish loss-free and duplication-free failure/recovery semantics by exploiting sequence numbers assigned to tuples. However, it currently considers only order-preserving or set-preserving operators and thus cannot support convergent-capable and divergent queries discussed in this paper.

9 Conclusion

In this paper, we argue that the distributed and data-flow nature of stream processing applications raises a number of interesting challenges and opportunities for high availability. We defined three types of recovery that provide increasingly stronger guarantees. We also defined four classes of operators and query networks that affect the results of applying certain types of recovery, making various guarantees more costly to achieve. Within this framework, we discussed how traditional high-availability approaches can be adapted to a DSPS and described a new stream-oriented approach that leverages the data-flow nature of stream processing.

Using analysis and simulations, we quantitatively characterized the runtime overhead and recovery time tradeoffs among the approaches. Process-pair based approaches, especially active standby, provide the fastest recovery but at a high cost. Passive standby is best suited to provide precise recovery for arbitrary query networks. The approach that we introduced, upstream backup, has a significantly lower runtime overhead but a longer recovery time that depends mostly on the size of the query-network state. The approach is best suited to an environment where failures are infrequent. 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.

We currently have a prototype implementation of amnesia recovery. In future work, we plan to extend our prototype and perform more experiments on real deployments. We also plan to investigate the issues of handling network partitions.

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*, Sep 2003.

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

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

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

[5] 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 VLDB*, Aug. 2002.

[6] S. Chandrasekaran, A. Deshpande, M. Franklin, and J. Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the 1st CIDR*, Jan. 2003.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the 1st CIDR*, 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, 1985.

[10] IBM Corporation. Getting the most out of MQSeries. White paper. http://www.bmc.com/resourcecenter/partners/mqseries/gettingthemostoutof%mqseries.html, 2003.

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

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

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

[14] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of the 18th ICDE*, 2002.

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

[16] 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 1st CIDR*, Jan. 2003.

[17] Oracle Inc. Oracle 10*g* high availability solutions. http://otn.oracle.com/deploy/availability.

[18] Oracle Inc. Oracle 9*i* streams - online documentation. http://www.oracle.com.

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

[20] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. An adaptive partitioning operator for continuous query systems. Technical Report CS-02-1205, UC. Berkeley, 2002.

[21] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the 2004 ACM SIGMOD*, June 2004.

[22] 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.