High-Availability Algorithms for Distributed Stream Processing [*]

Jeong-Hyon Hwang[†], Magdalena Balazinska[‡], Alexander Rasin[†],
Uğur Çetintemel[†], Michael Stonebraker[‡], and Stan Zdonik[†]

[†]Brown University   [‡]MIT

{jhhwang, alexr, ugur, sbz}@cs.brown.edu   {mbalazin, stonebraker}@lcs.mit.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 distributed environments. Unlike traditional data-processing applications that require precise recovery for correctness, many stream-processing applications can tolerate and benefit from weaker recovery guarantees. In this paper, we study various recovery guarantees and pertinent recovery techniques that can meet the correctness and performance requirements of stream-processing applications.

We discuss the design and algorithmic challenges associated with the proposed recovery techniques and describe how each can provide different guarantees with proper combinations of redundant processing, checkpointing, and remote logging. Using analysis and simulations, we quantify the cost of our recovery guarantees and examine the performance and applicability of the recovery techniques. We also analyze how the knowledge of query network properties can help decrease the cost of high availability.

1 Introduction

Stream-processing engines (SPEs) [1, 3, 5, 6, 16, 18] 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 monitoring (car traffic, air quality, battle field), financial applications (stock-price monitoring, ticker failure detection), and asset tracking. Because data sources are commonly located at remote sites, stream-based applications can gain in both scalability and efficiency if the 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, resulting in so-called distributed stream-processing systems (DSPSs) [6, 7, 22].

In a DSPS, the failure of a single server can significantly disrupt or even halt overall stream processing. Indeed, such

a failure causes the loss of a potentially large amount of transient information and, perhaps more importantly, prevents downstream servers from making progress. A DSPS therefore must incorporate a high-availability mechanism that allows processing to continue in spite of server failures. This aspect of stream processing, however, has received little attention until now [23]. In this paper, we focus on approaches where once a server fails, a backup server takes over the operation of the failed one. Tightly synchronizing a primary and a secondary so that they always have the same state incurs high run-time overhead. Hence, we explore approaches that relax this requirement, allowing the backup to *rebuild* the missing state instead.

Because different stream processing applications have different high-availability requirements, we define three types of recovery guarantees that address these different needs.

Precise recovery hides the effects of a failure perfectly, except for some transient increase in processing latency, and is well-suited for applications that require the post-failure output be identical to the output without failure. Many financial services applications have such strict correctness requirements.

Rollback recovery avoids information loss without guaranteeing precise recovery. The output produced after a failure is "equivalent" to that of an execution without failure, but not necessarily to the output of the execution that failed. The output may also contain duplicate tuples. To avoid information loss, the system must preserve all the necessary input data for the backup server to rebuild (from its current state) the primary's state at the moment of failure. Rollback recovery is thus appropriate for applications that cannot tolerate information loss but may tolerate imprecise output caused by the backup server reprocessing the input somewhat differently than the primary did. Example applications include those that alert when specific conditions occur (*e.g.*, fire alarms, theft prevention through asset tracking). We show in Section 6 that this recovery guarantee can be provided more efficiently than precise recovery both in terms of runtime overhead and recovery speed.

Gap recovery, our weakest recovery guarantee, addresses the needs of applications that operate solely on the most recent information (*e.g.*, sensor-based environment monitoring), where dropping some old data is tolerable for reduced recovery time and runtime overhead.

We define these recovery semantics more precisely in Sec-

tion 3. To the best of our knowledge, commercial DBMSs typically offer precise or gap recovery capabilities [8, 19, 20, 21] and no existing solution addresses rollback recovery or a similar weak recovery model.

We also investigate four recovery approaches that can provide one or more of the above recovery guarantees. Since each approach employs a different combination of redundant computation, checkpointing, and remote logging, they offer different tradeoffs between runtime overhead and recovery performance.

We first introduce *amnesia*, a lightweight scheme that provides gap recovery without any runtime overhead (Section 4). We then present *passive standby* and *active standby*, two process-pairs [4, 10] approaches tailored to stream processing. In passive standby, each primary server (a.k.a. node) periodically reflects its state updates to its secondary node. In active standby, the secondary nodes process all tuples in parallel with their primaries. We also propose *upstream backup*, an approach that significantly reduces runtime overhead compared to the standby approaches while trading off a small fraction of recovery speed. In this approach, upstream nodes act as backups for their downstream neighbors by preserving tuples in their output queues while their downstream neighbors process them. If a server fails, its upstream nodes replay the logged tuples on a recovery node. In Section 5, we describe the details of these approaches with an emphasis on the unique design challenges that arise in stream processing. Upstream backup and the standby approaches provide rollback recovery in their simplest forms and can be extended to provide precise recovery at a higher runtime cost, as we discuss in Section 6.

Interestingly, for a given high-availability approach, the overhead to achieve precise recovery can noticeably change with the properties of the operators constituting the query network. We thus develop in Section 3 a taxonomy of stream-processing operators, classifying them according to their impact on recovery semantics. Section 6 shows how such knowledge helps reduce high-availability costs and affects the choice of most appropriate high-availability technique.

Finally, by comparing the runtime overhead and recovery performance for each combination of recovery approach and guarantee (Section 7), we characterize the tradeoffs among the approaches and describe the scenarios when each is most appropriate. We find that upstream backup requires only a small fraction of the runtime cost of others, while keeping recovery time relatively short for queries with moderate state size. The size of query state and the frequency of high-availability tasks significantly influence the recovery performance of upstream backup and the runtime performance of passive standby. We also find that there is a fundamental tradeoff between recovery time and runtime overhead and that each approach covers a complementary portion of the solution space.

2 The System Model

A *data stream* is a sequence of tuples that are continuously generated in real time and need to be processed on arrival.
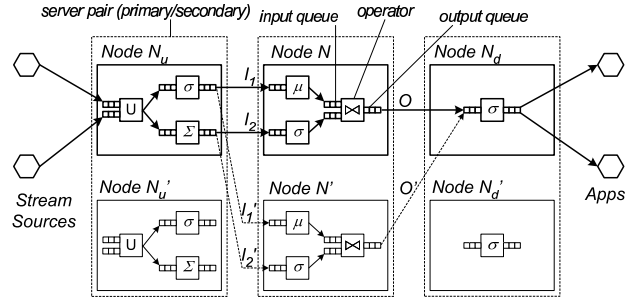

Figure 1. An example DSPS

This model of processing data before (or instead of) storing it contrasts with the traditional "process-after-store" model employed by all conventional DBMSs. In stream-processing systems [1, 3, 6], each *operator* is a processing unit (map, filter, join, aggregate, etc.) that receives input tuples through its *input queues* (one for each input stream) and produces output tuples based on its execution semantics. A loop-free, directed graph of operators is called a *query network*.

A DSPS partitions its query network across multiple nodes. Each node runs a stream-processing engine (SPE). Figure 1 illustrates a query network distributed across three nodes, N_u, N, and N_d. In the figure, streams are represented by solid line arrows while operators are represented as boxes labeled with symbols denoting their functions. Since messages flow on streams I_1 and I_2 from N_u to N, N_u is said to be *upstream* of N, and N is said to be *downstream* of N_u. We assume that the communication network ensures order-preserving, reliable message transport (*e.g.*, TCP).

Since we focus on *single-node fail-stop failures* (*i.e.*, handling network failures, partitions, or multiple simultaneous failures including those during recovery is beyond the scope of this paper), we associate each node N with a recovery node N' that is in charge of detecting as well as handling the failure of N. N in this case is called a *primary node*. For N' we use the terms *recovery node*, *secondary node*, and *backup node* interchangeably. Each recovery node runs its own SPE, and has the same query-network fragment as its primary, but its state is not necessarily the same as that of the primary.

To detect failures, each recovery node periodically sends keep-alive requests to its primary and assumes that the latter failed if a few consecutive responses do not return within a timeout period (for example, our prototype uses three messages with 100 ms transmission interval, for an average failure detection delay of 250 ms). When a recovery node detects the failure of its primary, if it was not already receiving the input streams, it asks the upstream nodes to start sending it the data (in Figure 1, I_1 and I_2 switch to I'_1 and I'_2 respectively). The recovery node also starts forwarding its output streams to downstream nodes (in Figure 1, O switches to O').

Because the secondary may need to reprocess some earlier input tuples to bring its state up-to-date with the pre-failure state of the primary, each redirected input stream must be able to replay earlier tuples. For this purpose, each output stream has an *output queue* as a temporary storage for tuples sent.

Finally, once a failed node comes back to life, it assumes the role of the secondary. As we discuss in Section 5, each approach requires a different amount of time for recovery and, thus, for the system to tolerate a new failure.

3 High-Availability Semantics

In this section, we define three recovery types, based on their effects as perceived by the nodes downstream from the failure. Since some operator properties facilitate stronger recovery guarantees, we also devise an operator classification based on their effects on recovery semantics.

3.1 Recovery Types

We assume that a query-network fragment, $\mathcal{Q}$, is given to a primary/secondary pair. $\mathcal{Q}$ has a set of n input streams $(I_1, I_2, ..., I_n)$ and produces one output stream O. The definitions below can easily be extended to query-network fragments with multiple output streams.

Because the processing may be non-deterministic, as we discuss in Section 3.2, executing $\mathcal{Q}$ over the same input streams may each time produce a different sequence of tuples on the output stream. We define an *execution* to be the sequence of events (such as the arrival, processing or production of a tuple) that occur while a node runs $\mathcal{Q}$. Given an execution e, we denote with O_e the output stream produced by e. We express the overall output stream after failure and recovery as $O_f + O'$, where is f the pre-failure execution of the primary and O' is the output stream produced by the secondary after it took over.

Precise Recovery: The strongest failure recovery guarantee, called *precise recovery*, completely masks a failure and ensures that the output produced by an execution with failure (and recovery) is identical to the output produced by an execution e *without failure*: i.e., $O_f + O' = O_e$.

Rollback Recovery: A weaker recovery guarantee, called *rollback recovery*, ensures that failures do not cause information loss. More specifically, it guarantees that the effects of all input tuples are always forwarded to downstream nodes in spite of failures. Achieving this guarantee requires:

1. *Input preservation* - The upstream nodes must store in their output queues all tuples that the secondary needs to rebuild, from its current state, the primary's state. We refer to such tuples as *duplicate input tuples* because they have already entered the primary node.

2. *Output preservation* - If a secondary is running ahead of its primary, the secondary must store tuples in its output queues until all the downstream nodes receive the corresponding tuples from the primary node. The tuples at the secondary are then considered *duplicate*.

Because the secondary may follow a different execution than its primary, duplicate output tuples are not necessarily identical to those produced by the primary. We consider an output tuple t at the secondary to be *duplicate* if the primary has already processed *all* input tuples that "affected" the value of t and forwarded the resulting output tuples downstream. We formally define rollback recovery and duplicate output tuples in [11].

Recovery type	Before failure			After failure			
-**Precise**	t_1	t_2	t_3	t_4	t_5	t_6	...
-**Gap**	t_1	t_2	t_3		t_5	t_6	...
-Rollback							
-**Repeating**	t_1	t_2	t_3	$\mathbf{t_2}$	$\mathbf{t_3}$	t_4	...
-**Convergent**	t_1	t_2	t_3	$\mathbf{t'_2}$	$\mathbf{t'_3}$	t_4	...
-**Divergent**	t_1	t_2	t_3	$\mathbf{t'_2}$	$\mathbf{t'_3}$	$\mathbf{t'_4}$	...

Figure 2. Outputs produced by each type of recovery

We use the configuration in Figure 1 to illustrate these concepts. We cannot discard tuples in the output queues of I_1 and I_2 if N' requires them to rebuild N's state. Similarly, if N' is running ahead of N, it must preserve all tuples in O''s output queue until they become duplicate (i.e., N_d receives from N tuples resulting from processing the same input tuples).

Rollback recovery allows the secondary to forward *duplicate output tuples* downstream. The characteristics of Q determine the characteristics of such duplicate output tuples as well as the properties of $O_f + O'$. We distinguish three types of rollback recovery. In the first type, *repeating recovery*, duplicate output tuples are *identical* to those produced previously by the primary. With the second type, *convergent recovery*, duplicate output tuples are different from those produced by the primary. The details on such situations are discussed in Section 3.2 under *convergent-capable* operators. In both recovery types, however, the concatenation of O_f and O' *after removing duplicate tuples* is identical to an output without failure, O_e. Finally, the third type of recovery, *divergent recovery*, has the same properties as convergent recovery regarding duplicate output tuples. Eliminating these duplicates, however, does not produce an output that is achievable without failure, because of the non-determinism in processing.

Gap Recovery: Any recovery technique that does not ensure both input and output preservation may result in information loss. This recovery type is called *gap recovery*.

Example: Figure 2 shows examples of outputs produced by each recovery type. With precise recovery, the output corresponds to an output without failure: tuples t_1 through t_6 are produced in sequence. With gap recovery, the failure causes the loss of tuple t_4. Repeating recovery produces tuples t_2 and t_3 twice. Convergent recovery generates different tuples t'_2 and t'_3 after failure (but corresponding to t_2 and t_3) but then produces tuples t_4 and following as would an execution without failure. Finally, divergent recovery keeps producing equivalent rather than identical tuples after the failure.

Propagation of Recovery Effects: The semantics above define the effects of failure and recovery on the output stream of the failed query-network fragment. These effects then propagate through the rest of the query network until they reach client applications. Because precise recovery masks failures, no side effects propagate. Gap recovery may lose tuples. After a failure, client applications may thus miss a burst of tuples. Because the query network may aggregate many tuples into a single output tuple, missing tuples may also result in incorrect output values: e.g., a sum operator may produce a lower sum. Rollback recovery does not lose tuples but may generate duplicate tuples. The final output stream may thus
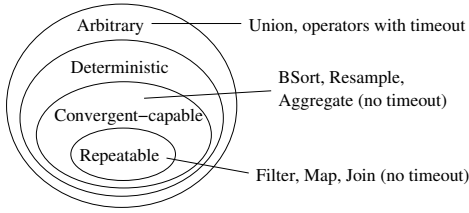
Figure 3. Taxonomy of Aurora operators

p	per-input-tuple processing time
d	network transmission delay between any nodes
λ	input tuple arrival rate
C	size of checkpoint message
c	size of queue-trimming message
M	checkpoint or queue-trimming interval
D	failure detection delay
r	time to redirect input streams
nb_ops	number of operators in the query network
nb_paths	number of paths from input to output streams
Δ	number of lost or redundant tuples
K	delay before processing first duplicate input tuple
Q	average number of input tuples to re-process
rec_time	time spent recreating the failed state (after failure detection)
bw_overhead	$\frac{\text{bandwidth consumed for high availability}}{\text{bandwidth consumed for tuple transmission}}$
proc_overhead	$\frac{\text{processing required for high availability}}{\text{processing required for ordinary tuple processing}}$

Table 1. Summary of notation

contain a burst of either redundant or incorrect tuples: *e.g.*, a sum operator downstream may produce a higher sum value. It is also possible, however, that duplicate-insensitive operators (*e.g.*, max) downstream can always guarantee correct results. In general, the recovery type for a node must be chosen based on the applications' correctness criteria and the characteristics of the operators on the node and downstream.

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 classification of Aurora operators [1, 2]. The type of a query network is determined by the type of its most general operator.

An operator is *deterministic* if it produces the same output stream 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, which interleaves tuples from multiple streams), and use of non-determinism in processing such as randomization.

A deterministic operator is called *convergent-capable* if it yields a convergent recovery when it restarts from an empty internal state and re-processes the same input streams, starting from an arbitrary earlier point in time. 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 execution that would have existed without failure. Window alignment is the only possible cause that prevents a deterministic operator from being convergent-capable. This is because window boundaries define the sequences of tuples over which operators perform computations. 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 it yields a repeating recovery when it restarts from an empty internal state and re-processes the same input streams, starting from an arbitrary earlier point in time (the operator must produce identical duplicate 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 thus not repeatable in general, whereas filter (which simply drops tuples that do not match a given predicate) and map (which transforms tuples by applying functions to their attributes) are repeatable as they have one input stream and process each tuple independently of others. Join (without timeout) is also repeatable because its windows defined on input streams have alignments relative to the latest input tuple being processed.

In the following sections, we present approaches for 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 tradeoffs between recovery time and runtime overhead. Table 1 summarizes the notation that we use.

4   Gap Recovery

The simplest approach to high availability is for the secondary node to restart the failed query network from an empty state and continue processing input tuples as they arrive. This approach, called *amnesia*, produces a gap recovery for all types of query networks. In amnesia, 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 imposes no overhead at runtime (c.f. Table 3).

We define *recovery time* as the interval between the time when the secondary discovers that its primary failed and the time it reaches the primary's pre-failure state (or an equivalent state for a non-deterministic query network). Recovery time thus measures the time spent recreating the failed state.

Since amnesia does not recreate the lost state and drops tuples until the secondary is ready to accept them, the recovery time is zero. It takes time r to redirect the inputs to the secondary, but when processing restarts, the first tuples processed are those that would have been processed at the same time if the failure did not happen. *I.e.*, there is no extra delay due to the failure or recovery.

5   Rollback Recovery Protocols

We present three approaches to achieve rollback recovery, each one using a different combination of redundant computation, checkpointing, and logging at other nodes. We first

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

	rec_time	bw_overhead	proc_overhead
Amnesia	0	0	0
Passive standby	$K + Qp$, where $K = r + d$; $Q = \frac{M\lambda}{2}$	$f_1(\frac{1}{M}, C)$	$f_2(\frac{1}{M}, C)$
Upstream backup	$K + Qp$, where $K = r + d$; $Q = |\text{state}| + M\lambda + 2d\lambda$	$f_3(\frac{1}{M}, c)$	$f_4(\frac{1}{M}, \text{nb\_ops}, \text{nb\_paths})$
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

present *passive standby*, an adaptation of the process-pairs model with passive backup. Passive standby relies on checkpointing to achieve high availability. Then, we introduce *upstream backup*, where upstream nodes in the processing flow serve as backup for their downstream neighbors by logging their output tuples. Finally, we describe *active standby*, another adaptation of the process-pairs model where each standby performs processing in parallel with its primary. We discuss active standby last, because it relies on concepts introduced in upstream backup.

For each approach, we examine the recovery guarantees it provides, the average recovery time, and the runtime overhead. We divide the runtime overhead into processing and communication (or bandwidth) overhead. Table 2 summarizes the recovery types achieved by each approach while Table 3 summarizes their performance metrics.

5.1 Passive Standby

In passive standby, each primary periodically sends the delta of its state to the secondary, which takes over from the latest checkpoint when the primary fails. Since real-time response is crucial for many stream-processing applications, the main challenge in passive standby is to enable the primary to continue processing even during a checkpoint.

The state of a query network consists of the states of input queues of operators, operators themselves, and the node output queues (one for each output stream). Each checkpoint message (a.k.a. state update message) thus captures the changes to the states of those queues and operators on the primary node since the last checkpoint message was composed. For each queue, the checkpoint message contains the newly enqueued tuples as well as the last dequeue position. For an operator, however, the content of the message depends on the operator type. For example, the message is empty for stateless operators while it stores, for an aggregate operator, either some summary values (e.g., count, sum, etc.) or the actual tuples that newly entered the operator's state.

To avoid the suspension of processing, the composition of a checkpoint message is conducted along a virtual "sweep line" that moves from left (upstream) to right (downstream). At every step, an operator closest to the right of the sweep line is chosen and once its state difference is saved in the checkpoint message, the sweep line moves to the right of the operator. The primary is free to execute operators away from the

sweep line both upstream and downstream because these concurrent tasks do not violate the consistency of the checkpoint message. Indeed, executing operators to the left of the sweep line is equivalent to executing them after checkpointing. Executing operators to the right of the sweep line corresponds to executing them before the message composition.

Passive standby guarantees rollback recovery as follows: (1) *input preservation* - each upstream primary node preserves output tuples in its output queues until they are safely stored at the downstream secondaries. In Figure 1, whenever standby node N' receives a checkpoint from N, it informs upstream node N_u about the new tuples that it received on its input streams, I'_1 and I'_2. N_u discards only those acknowledged tuples from its output queues. (2) *output preservation* - the secondary is always "behind" the primary because its state corresponds to the last checkpointed sate.

If a primary fails, the secondary takes over and sends all tuples from its output queues to the downstream nodes. The secondary also asks upstream nodes to start sending it their output streams, including tuples stored in their output queues. When the failed node rejoins the system, it assumes the role of the secondary. Because the new secondary has an empty state, the primary sends its complete state in the first checkpoint message.

Recovery Type: Because the secondary node restarts from a past state of its primary, passive standby provides repeating recovery for deterministic query networks and divergent recovery for others.

Recovery Time: Passive standby has a short recovery time because the backup holds a complete and recent snapshot of the primary's state. Recovery time is equal to $K + Qp$, where K is the delay before the recovery node receives its first input tuple, Q is the number of duplicate input tuples it reprocesses, and p is the average processing time per input tuple. K is the sum of r (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. The average number, Δ, of duplicate tuples is close to $M\lambda_{out}$, where M is the checkpoint interval and λ_{out} is the rate of tuples on output streams.

Overhead: Passive standby imposes high runtime overhead. The bandwidth overhead is inversely proportional to
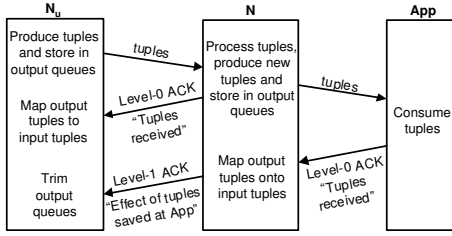
Figure 4. Inter-node communication in upstream backup

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) determines the tradeoff between runtime overhead and recovery time. Table 3 summarizes these results.

5.2 Upstream Backup

In upstream backup, *upstream nodes act as backups for their downstream neighbors* by logging tuples in their output queues until all downstream neighbors completely process these tuples. For instance, in Figure 1, node N_u serves as backup for node N: if N fails, N' restores the lost state by re-processing the tuples logged at N_u. When a failed node rejoins the system, it assumes the role of the recovery node starting from an empty state. The system is then able to tolerate a new failure without further delay.

The main difficulty of this approach is to determine the maximum set of logged tuples that can safely be discarded given operator non-determinism and the many-to-many relationship between input and output tuples.

Figure 4 shows a typical communication sequence between three nodes N_u, N, and *App*. Each node produces and sends tuples downstream while storing them in its output queues. Each node also periodically acknowledges reception of input tuples by sending level-0 acks to its direct upstream neighbors. When a node (*e.g.*, N) receives level-0 acks from downstream neighbors (*e.g.*, *App*), it notifies its own upstream neighbors (*e.g.*, N_u) about the earliest logged tuples (one per N_u's output) that contributed to producing the acknowledged tuples and are thus the oldest tuples necessary to re-build the current state (of N). Discarding only earlier tuples allows the system to survive single failures. The notifications are thus called level-1 acks (denoted $\text{ACK}(1, S, u)$, where S identifies a stream and u identifies a tuple on that stream). Leaf nodes in the DSPS use level-0 acks to trim their output queues.

Since upstream nodes log all tuples necessary for the secondary to re-build the primary's state from an empty state (input preservation) and the secondary restarts from an empty state (output preservation), upstream backup provides rollback recovery.

5.2.1 Queue Trimming Protocol

To avoid spurious transmissions, nodes produce both level-0 and level-1 acks every M seconds. A lower ack frequency
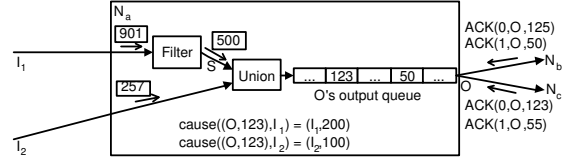
(a) N_a receives acks from downstream and new tuples from upstream. The filter processes $I_1[900]$ and produces $S[500]$
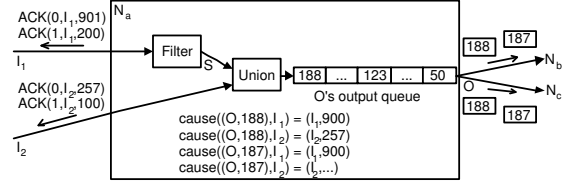
(b) N_a trims its output queue at $(O, 50)$ while pushing new tuples $O[187]$ and $O[188]$ downstream. N_a also maps the lowest level-0 ack received, $(O, 123)$, onto level-1 acks

Figure 5. One iteration of upstream backup

reduces bandwidth utilization, but increases the size of output queues and the recovery time.

To compose level-1 acks, each node finds, for each output stream O, the latest output tuple $O[v]$ acknowledged at level-0 by all downstream neighbors. For each input stream I, the node maps $O[v]$ back onto the earliest input tuple $I[u]$ that caused $O[v]$. This backward mapping is conducted by a function $\text{cause}((O, v), I) \to (I, u)$, where (I, u) denotes the identifier of tuple $I[u]$ and marks the beginning of the sequence of tuples on I necessary to regenerate $O[v]$. We discuss the cause function next. The node performs these mappings for each output stream and identifies the earliest tuple on each input stream that can now be trimmed. The node produces level-1 acks for these tuples. Each upstream neighbor trims its output queues up to the position that corresponds to the oldest tuple acknowledged at level-1 by all downstream neighbors. We present this algorithm in more detail in [11].

Figure 5 illustrates one iteration of the upstream-backup algorithms on one node. In the example, node N_a receives level-0 and level-1 acks from two downstream neighbors N_b and N_c. First, since both neighbors have now sent level-1 acks for tuples up to $O[50]$, N_a removes from its output queue all tuples preceding $O[50]$. Second, since both N_b and N_c have sent level-0 acks for tuples up to $O[123]$, N_a maps $O[123]$ back onto the first input tuples that caused it. N_a sends level-1 acks for these tuples, identified with $(I_1, 200)$ and $(I_2, 100)$. In the example, N_a also receives tuples $I_1[901]$ and $I_2[257]$ from its upstream neighbors and acknowledges their reception with level-0 acks.

5.2.2 Mapping Output Tuples onto Input Tuples

We now discuss how nodes compute the cause function, $\text{cause}((O, v), I) \to (I, u)$. This function maps an arbitrary

output tuple $O[v]$ on stream O onto the earliest input tuple $I[u]$ on input stream I that has contributed to the production of $O[v]$ (*i.e.* affected the value of $O[v]$). To facilitate this mapping, we propose to keep track of the oldest input tuples that affect any computation, by appending *input-tuple indicators* to tuples as they travel through operators on a node. For a tuple $O[v]$, these indicators, denoted with $\mathrm{indicators}(O, v)$, contain the identifiers of the oldest tuples on input streams necessary to generate $O[v]$. We also call these indicators *low watermarks*. On any stream, indicator values are monotonically non-decreasing.

When a tuple enters a node, its indicators are initialized to its identifier: *e.g.,* $\mathrm{indicators}(I, u) = \{(I, u)\}$. Each operator uses the indicators of its input tuples to compute the indicators for its output tuples. When it is first set up, each operator o initializes a watermark variable ω for each node-wide input stream I that contributes to each input stream S of o: $\omega[I, S] = 0$. As it processes tuples, the operator updates each $\omega[I, S]$ to hold the indicator of the oldest tuple currently in the state or, for stateless operators, the indicators of the last tuples processed. When it produces a tuple t, the operator iterates through all ω values and appends (I, ω_{min}) to $\mathrm{indicators}(t)$, where ω_{min} is the minimum of all $\omega[I, *]$.

Some operators, such as union, have many input streams but only a few of them actually contribute to any single output tuple. These operators can reduce the number of indicators on output tuples by appending only indicators for input streams that actually affected the output tuple value. Thus, $\mathrm{cause}((O, v), I)$ refers to the indicator of $O[v]$ that corresponds to stream I, or to the indicator of the last preceding tuple affected by I, if $O[v]$ was not affected by I. Note that indicators are not sent to downstream nodes. More details about the use of indicators can be found in [11].

Figure 5 shows an example of managing input-tuple indicators. In Figure 5(a), the filter processes $I_1[900]$ and produces $S[500]$. Hence, $\mathrm{indicators}(S, 500) = \{(I_1, 900)\}$. In Figure 5(b), the union operator processes tuples $S[500]$ and $I_2[257]$ to produce $O[187]$ and $O[188]$ respectively. Hence, $\mathrm{indicators}(O, 187) = \{(I_1, 900)\}$ and $\mathrm{indicators}(O, 188) = \{(I_2, 257)\}$. Therefore, $\mathrm{cause}((O, 188), I_1) = (I_1, 900)$, $\mathrm{cause}((O, 188), I_2) = (I_2, 257)$, and $\mathrm{cause}((O, 187), I_1) = (I_1, 900)$. $\mathrm{cause}((O, 187), I_2)$ depends on the indicators of the tuples preceding O.

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 the standby approaches.

Recovery Time: The time, K, to receive the first tuple is the same as for passive standby but the recovery node may re-process significantly more tuples. It must re-process (1) all tuples that contributed to the lost state, (2) a complete queue-trimming interval worth of tuples on average (due to the periodic transmission of both level-0 and level-1 acks), and (3) some extra tuples that account for the propagation delays of level-0 acks. 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 bandwidth overhead because queue-trimming messages, which contain only the tuple identifiers for streams crossing node boundaries, are significantly smaller than checkpoint messages used by the other approaches. 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

Active standby is another variation on the process-pairs model. In contrast to passive standby, with active standby, each secondary node receives tuples from upstream and processes them in parallel with the primary. The secondary, however, does not send any output tuples downstream. It logs these tuples in its output queues instead.

The challenge of active standby lies in bounding the output queues on each secondary, while ensuring output preservation. Because the primary and secondary may have non-deterministic operators, they may have different tuples in their output queues. To identify duplicate output tuples, we add a second set of input-tuple indicators to each tuple. For a tuple, $O[v]$, this second set contains for each input stream I, the identifier (I, u) of the *most recent* tuple that contributed to the production of $O[v]$. We call these identifiers *high watermarks*. A tuple at the secondary is duplicate if it has a lower-valued high watermark than a tuple at the primary. Indeed, this tuple results from processing the same or even older input tuples. Each secondary thus trims all logged output tuples that have a high watermark lower than the high watermarks of the tuples already received by downstream nodes. For high watermarks to be correct, we need to distinguish input-tuple indicators that travel on different paths through a node. We discuss these details further in [11].

Watermarks are never sent between upstream and downstream nodes but they are sent between primary and secondary nodes, as illustrated in the following example. We use Figure 5 to illustrate active standby but we assume indicators are high watermarks. When $\mathrm{ACK}(0, O, 125)$ and $\mathrm{ACK}(0, O, 123)$ arrive, node N_a determines that $O[123]$ is now acknowledged at level-0 by both downstream neighbors. Since tuple $O[123]$ maps onto input tuples identified with $(I_1, 200)$ and $(I_2, 100)$, the set of identifiers $\{(I_1, 200), (I_2, 100)\}$ is added to the queue-trimming message as the entry value for O. When the secondary receives the queue-trimming message, it discards tuples u (from the output queue corresponding to O) for which $\mathrm{cause}((O, u), I_1)$ returns a tuple older than $I_1[200]$ and $\mathrm{cause}((O, u), I_2)$ returns a tuple older than $I_2[100]$.

If the primary fails, the secondary takes over by sending the logged tuples to all downstream neighbors, and then con-

	Passive standby		
Q. network	bw_overhead	proc_overhead	rec_time
Deterministic	none	negligible	none
Arbitrary	none	negligible	none
	Active standby		
Q. network	bw_overhead	proc_overhead	rec_time
Deterministic	none	negligible	r
Arbitrary	determinants	determinants	$r + f_5$(log. freq.)
	Upstream backup		
Q. network	bw_overhead	proc_overhead	rec_time
Repeatable	$\frac{f(k)*\text{size}(\texttt{tuple\_id})}{\text{size}(\texttt{tuple})}$	negligible	none
Convergent	$\frac{f(k)*\text{size}(\texttt{tuple\_id})}{\text{size}(\texttt{tuple})}$	double	negligible
Arbitrary	determinants	determinants	negligible

Table 4. Added overhead for precise recovery

tinuing its processing. When the failed node rejoins the system as the new secondary, it starts with an empty state and becomes up-to-date with respect to the new primary only after processing sufficiently many input tuples. Active standby guarantees rollback recovery since each secondary always receives what its primary receives (*input preservation*) and each secondary discards logged output tuples only when they become duplicate (*output preservation*).

Recovery Type: Because the secondary processes tuples in parallel with the primary, active standby provides repeating recovery for all deterministic query networks and divergent recovery for others.

Recovery Time: Because the standby continues processing during failure, it only needs to transmit all duplicate 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. M determines the trimming interval for the secondary's output queues.

Overhead: Because all processing is replicated by the standby node, both proc_overhead and bw_overhead are approximately 100%. The overheads are actually somewhat higher due to the processing of input-tuple indicators and transmitting queue-trimming messages. Table 3 summarizes these results.

6 Precise-Recovery Extensions

All recovery approaches can achieve precise recovery for convergent-capable query networks, by eliminating duplicate 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: Passive standby provides repeating recovery for deterministic query networks. 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 then discard all tuples preceding the ones identified. These requests can be made while the recovery node regenerates the failed state, achieving precise recovery without additional overhead. For a non-deterministic query network, because the secondary may produce different duplicate output tuples when it takes over, the primary

can only forward checkpointed tuples downstream. This constraint causes bursty output while also increasing the end-to-end latency.

Active Standby: For a deterministic query network, active standby also makes recovery precise by asking downstream nodes for the identifiers of the latest tuples they received. The delay imposed by this request cannot be masked and thus extends the recovery time by r. For other query networks, we must ensure that both the primary and secondary follow the same execution. To do so, whenever a non-deterministic operator executes, the primary must collect all information necessary to replay the execution of the operator. The primary accumulates such information, called determinants [9][1], in a log message. Determinants affect both bandwidth and processing overhead. The logging frequency affects (1) the recovery time, because non-deterministic operators on the secondary cannot execute until they obtain appropriate determinants, and (2) the end-to-end delay, because the primary cannot send tuples downstream until the secondary receives all determinants involved in generating these tuples.

Upstream Backup: In repeatable query networks, operators produce output tuples by combining at most one tuple from each input stream. Input-tuple indicators therefore uniquely identify tuples and can serve for duplicate elimination, offering precise recovery with negligible extra processing overhead. For a convergent query network, the secondary must be able to remove duplicate output tuples during recovery. It achieves this by using the additional high watermarks as discussed in Section 5.3. This approach thus doubles the processing overhead. For repeatable query networks, nodes forward low watermarks downstream while for convergent-capable query networks, nodes forward high watermarks instead. In both cases, the extra bandwidth overhead is approximately $\frac{f(k)*\text{size}(\text{tuple\_id})}{\text{size}(\text{tuple})}$, where $f(k)$ is a function of the average number of input streams (at a node) that contribute to an output stream. As in active standby, upstream backup can provide precise recovery for more complex query networks by logging determinants from primary to secondary. Unlike active standby, these determinants are processed only when the secondary takes over. The details of the protocol are presented in [11].
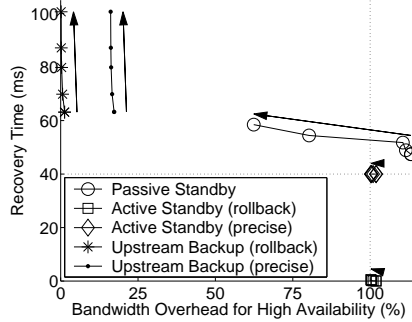
7 Evaluation

We evaluate and compare the performance of each approach through simulations. Using CSIM [17], we built a detailed simulator of a DSPS. Table 5 summarizes the main simulation parameters. The parameter values were obtained from our prototype implementation, which currently supports all our recovery types for simple repeatable query networks. Each point shown in the figures is the average of 25 simulation runs, at least one simulated minute each. Because amnesia has no overhead and a zero recovery time, but provides

---

[1]The representation of a determinant depends on the operator type. For example, the determinant for a random filter could be represented as a bit vector where each bit shows whether the corresponding tuple passed or was dropped. For a union operator, the determinant must include the exact interleaving of tuples.

Parameter	Meaning	Default
λ	input tuple arrival rate (tuples/s)	1000
D	delay to detect the failure of a node (ms)	250
M	queue-trimming/checkpoint interval (ms)	50
r	time to redirect input streams (ms)	40
Tuple	size of a tuple and size of a tuple id (bytes)	50, 8
Network	bandwidth(Mbps) and delay(ms)	16, 5
Proc. Cost	avg. processing time per input tuple (μs)	
Filter	—	10
Aggregate	(Proc. Cost of Filter) $* \text{Window} * \frac{1}{\text{Advance}}$	100
Selectivity	expected value of $\frac{\text{\# of output tuples emitted}}{\text{\# of input tuples consumed}}$	0.1

Table 5. Simulation parameters and their default values

Figure 6. Recovery time and runtime overhead for rollback and precise recovery as the communication interval varies from 25 ms to 200 ms (indicated by the arrows)

only gap recovery, we focus our evaluation on the other three approaches.

We first examine the overhead and recovery performance of each approach for rollback recovery and a convergent-capable query network (Section 7.1). We then evaluate the added overhead of achieving precise recovery (Section 7.2) and examine the effect of query-network types and other query-network properties on the performance of each approach (Section 7.3). We finally examine how performance changes as a function of query network size (Section 7.4).

For the overhead measurements, we only present bandwidth overhead because processing overhead poses similar tradeoffs while being more difficult to reproduce and evaluate accurately in simulations. We refer the reader to Sections 4 through 6 for a detailed discussion of processing overheads.

7.1 Runtime Overhead vs Recovery Time

To examine the runtime overhead and recovery time tradeoffs for rollback recovery and a convergent-capable query network, we simulate an aggregate with 100 ms window, 10 ms advance (this aggregate consumes 10% of a node's processing capacity) and default values for other parameters.

The only tunable parameter for each approach is the communication interval, which is the queue-trimming interval for upstream backup and active standby and the checkpointing interval for passive standby. Figure 6 shows the relation between recovery time and bandwidth overhead as the communication interval varies from 25, to 50, 100, 150, and 200 ms.

Looking at the overhead, upstream backup is the clear winner with an overhead close to zero. Even with a 25 ms communication interval, the node transmits only one 8-byte tuple identifier for every 25 tuples it receives, yielding an overhead of 0.64%. Upstream backup, however, has the slowest recovery as it must recreate the complete state of the failed query network. Upstream backup's recovery time is also most sensitive to the duration of the communication interval. Frequent trimming reduces recovery time for a negligible added overhead until the size of the query network and the time to redirect the input streams (r is 40 ms in our prototype) eventually limits the recovery speed. Recovery time is still relatively short compared with the 250 ms failure detection delay.

Active standby has an overhead of at least 100% because the secondary receives all input tuples in parallel with the primary. Queue-trimming messages used to discard output tuples from the secondary make the overhead slightly exceed 100%. Active standby has a negligible recovery time, though. The secondary only needs to resend half a queue-trimming interval worth of duplicate tuples stored in its output queues.

Passive standby's recovery time is between that of the other approaches because the secondary already has a snapshot of the last checkpoint but must ask upstream nodes to redirect their output streams and must re-process on average half a checkpoint worth of tuples. Passive standby's overhead varies significantly with the communication interval as each checkpoint message contains an update of the query-network state. 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 100 ms corresponds to the 100 ms window size. The curve would be smoother for a larger query network.

7.2 Cost of Precise Recovery

Figure 6 also presents the recovery time and runtime overhead of precise recovery. For passive standby and active standby, precise recovery of convergent-capable query networks adds no runtime overhead compared with rollback recovery. Precise recovery increases the runtime overhead of upstream backup by a little over 16% (equal to: $\frac{k*\text{size(tuple\_id)}}{\text{size(tuple)}}$, with $k = 1$, and $\frac{\text{size(tuple\_id)}}{\text{size(tuple)}} = \frac{8}{50} = 0.16$) because watermarks are now sent downstream. The overhead thus remains much lower than that of the process-pair based approaches.

For upstream backup and passive standby, 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. 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). Active standby cannot mask this delay and recovery extends by the constant value r (40 ms in our prototype). Overall, all approaches can offer precise recovery for convergent-capable query networks at a negligible incremental cost.

Query Net-work Type	Result	Upstream Backup	Active Standby	Passive Standby
Repeatable	Bw overhead (%)	0.64	100.96	101.27
	Rec. time (ms)	47.62	1.80	45.88
Convergent-capable	Bw overhead	0.64	100.96	**111.55**
	Rec. time	**69.86**	0.07	48.88
Non-deterministic	Bw overhead	1.28	101.91	101.90
	Rec. time	50.92	1.82	47.24

Table 6. Effects of query-network type

7.3 Effects of Query-Network Type

We now 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 merges two streams (500 tuples/s each) into one. Interestingly, the results show that neither the overheads nor the recovery times of the approaches are affected by the query network *type*.

Upstream backup and active standby use queue-trimming messages. Their overheads thus depend on the relative rates of these messages and tuples on input streams rather than any other property of the query network. In Table 6, the union has a slightly 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. Because the aggregate has the greatest differences in state between checkpoints, its overhead is highest with passive standby.

Active standby recovers by retransmitting output tuples. In Table 6, the output rate is ten times lower for the aggregate because of the 10 ms advance, resulting in a faster recovery for that operator. 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 re-process depends on the size of the query-network state. For these reasons, the aggregate has the longest recovery time with these approaches, especially with upstream backup. For passive standby the increase is negligible compared with the stream redirection delay.

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

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

Window size (tuples)	100	200	300	400	500
PS overhead (%)	111.55	111.55	111.54	111.54	111.54
PS rec. time (ms)	48.9	51.7	54.6	60.0	63.9
UB rec. time (ms)	69.9	98.9	138.7	188.5	248.3

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 (ms)	47.5	47.5	47.6	48.8	51.6
UB rec. time (ms)	62.6	61.4	61.3	69.9	83.8

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

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 112%. The recovery time of passive standby due to reprocessing tuples (the part in excess of 40 ms) 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 time spent reprocessing tuples increases roughly linearly with the size of the state. Upstream backup must indeed reprocess 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 increasing the rate at which the state of a query network changes using an aggregate operator with decreasing window advance from 100 ms to 5 ms and thus increasing selectivity from 0.01 to 0.2. Table 8 shows the impact of this increase in query-network state-update 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 with the magnitude of changes in query-network state. The advance determines the number of tuples that the operator produces during a checkpoint interval. This number increases from 1 to 20 as the advance decreases from 100 to 5 ms.

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 tuples 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 10 ms, the recovery time starts increasing.

In summary, for rollback recovery, the size of the query-
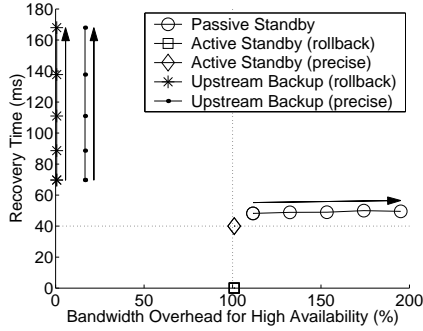
Figure 7. Effects of the number of operators. The arrows indicate the directions of the trends

network state increases upstream backup's recovery time while the rate and magnitude at which that state changes impacts the runtime overhead of passive standby.

7.4 Effect of Network Size

Increasing the size and complexity of the query network translates into increasing the size of the query-network state, the rate at which this state changes, and the processing complexity. As an example, Figure 7 shows the performance of each approach for a chain of 1 to 5 aggregate operators (with the parameter values from 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 because the processing complexity of each tuple increases. The recovery time of upstream backup increases rapidly as the state of the query network increases with each extra aggregate. It reaches 170 ms for 5 operators, which is still relatively short compared with the 250 ms failure detection delay. 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 results show that each approach poses a clear trade-off between recovery time and processing overhead. Active standby, with its high overhead and negligible recovery time, appears particularly well suited for systems where quick recovery justifies high runtime costs (*e.g.*, financial services, military applications).

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. It is thus best suited for applications, such as patient monitoring and other medical applications, that impose a somewhat lower load on the system but necessitate precise recovery.

Additionally, both in our prototype and simulator, we make the first nodes in the system adopt the passive-standby model since other approaches impose extra requirements on stream sources. Active standby requires that each source sends the stream to two different locations and upstream backup requires that each source loggs the tuples it produces.

Upstream backup provides precise recovery for most query networks with the 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 short recovery delays are tolerable and is thus particularly suitable for sensor-based environment and infrastructure monitoring applications. In contrast to process-pair approaches, recovery nodes can be chosen among live nodes allowing all servers to process data streams at runtime.

8 Related Work

Reliability through redundant processing, checkpointing, and logging has been widely studied in the context of traditional applications [9]. Recently, there has been much work on data-stream processing (e.g., Aurora [1, 5], STREAM [18], TelegraphCQ [6]), including proposals for distributed engines [7, 22]. In this paper, we investigate how to achieve high availability in these new systems.

The process-pairs model is adopted by many existing DBMSs [8, 19, 21, 20]. Oracle10g/DataGuard [19] is one such facility built on top of Oracle Streams [20]. The latter enables cross-database event propagation and trigger-rule-based processing of event streams. DataGuard supports three recovery modes: maximum protection (MPR), availability (MAV), and performance (MPE). MPR synchronously applies the same update to multiple machines as part of the same transaction, providing precise recovery. MPE asynchronously transmits redo logs to the standby, providing gap recovery only. MAV switches between MPR and MPE based on the accessibility of the standby. Our approaches provide precise recovery at a lower overhead because checkpoints are asynchronous and they also offer rollback recovery.

Commercial workflow systems [13] also rely on redundant components to achieve high availability. A variation of the process-pairs approach is used in the Exotica workflow system [14]. 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 [15], which efficiently resumes failed warehouse loads, is also 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 across a set of parallel servers [12, 22]. 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 [12], messages that are being processed by a server when the failure happens are trapped until the server recovers. Flux [23] introduces a 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. It currently only considers order-preserving or set-preserving operators though 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 novel challenges and opportunities for high availability. We define three recovery types that provide increasingly stronger guarantees. We also define four classes of operators and query networks based on their impact on the cost of providing various recovery guarantees. Within this framework, we introduce three recovery approaches that provide the proposed guarantees with different combinations of redundant processing, checkpointing, and logging.

Using analysis and simulations, we quantitatively characterize the runtime overhead and recovery time tradeoffs among the approaches. We find that each approach covers a complementary portion of the solution space. Process-pair based approaches, especially active standby, provide the fastest recovery but at a high cost. Active standby is thus best suited for environments where fast failure recovery (i.e., minimal disruptions) justifies higher runtime costs. Passive standby is best suited to provide precise recovery for arbitrary query networks. In contrast, upstream backup has a significantly lower runtime overhead but a longer recovery time that depends mostly on the size of the query-network state. This approach is thus best suited for an environment where failures are infrequent and short recovery delays are tolerable.

We currently have a basic prototype implementation that can provide the proposed recovery types for repeatable query networks. We will extend our prototype to support arbitrary query networks and perform experiments on real deployments. We also plan to investigate how to simultaneously use different recovery approaches at nodes in a DSPS, and, thus leverage the benefits of all schemes. We also plan to study network partitions, multiple failures, and the interaction between high availability and load balancing.

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

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

[11] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. Technical Report CS-04-05, Department of Computer Science, Brown University, 2004.

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

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

[14] 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.

[15] 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.

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

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

[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 1st CIDR*, Jan. 2003.

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

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

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

[22] 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.

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