A Cooperative, Self-Configuring High-Availability Solution for Stream Processing*

Jeong-Hyon Hwang, Ying Xing, Uğur Çetintemel, and Stan Zdonik
Department of Computer Science, Brown University
{jhhwang, yx, ugur, sbz}@cs.brown.edu

Abstract

We present a collaborative, self-configuring high availability (HA) approach for stream processing that enables low-latency failure recovery while incurring small run-time overhead. Our approach relies on a novel fine-grained checkpointing model that allows query fragments at each server to be backed up at multiple other servers and recovered collectively (in parallel) when there is a failure.

In this paper, we first address the problem of determining the appropriate query fragments at each server. We then discuss, for each fragment, which server to use as its backup as well as the proper checkpoint schedule. We also introduce and analyze operator-specific delta-checkpointing techniques to reduce the overall HA cost. Finally, we quantify the benefits of our approach using results from our prototype implementation and a detailed simulator.

1 Introduction

Recently, there has been significant interest in a new class of applications where high-volume, continuous data streams need to be processed with low latency. Such applications include sensor-based patient monitoring, asset tracking, traffic monitoring, real-time stock analysis, battlefield monitoring, etc. Since these applications monitor real-time events, the value of a result decays rapidly over time. Thus, processing latency is a significant issue.

As a response to these applications, several stream processing research prototypes [6, 4, 7] and commercial products have appeared. In these systems, processing is typically expressed as a dataflow graph of operators. The streaming input data passes through these operators while being transformed on its way to the outputs. This model of processing data before (or instead of) storing it sharply contrasts with the traditional "process-after-store" model employed in conventional database management systems (DBMSs). There has also been recent effort in the area to enhance system scalability by use of inexpensive, commodity clusters [9, 20, 24, 23].

In a distributed stream processing system (DSPS), deploying more servers in general improves the system performance. However, it also increases the risk of failure. In this setting, a failure indeed has a serious negative impact as it blocks the processing that should take place at the failed server. The intrinsic real-time nature of stream processing necessitates a high-availability (HA) solution that enables *fast recovery* and *minimal slow-down of regular processing*.

In this paper, we introduce a novel self-configuring HA approach that significantly outperforms existing ones during both recovery and normal processing. Whereas the previous approaches [19, 13, 5] mask a server failure by failing over to a standby server, our solution enables a set of servers to collectively take over the failed execution, realizing significantly faster recovery. Our solution also has low negative impact on regular processing as it strives to use idle cycles to execute short-duration HA tasks.

To maintain the backups spread over multiple machines, we use *checkpointing*, which incrementally copies any change in state to the backups. This is because checkpointing effectively works for a larger set of workload and usage cases than other alternatives that are based on either replay or redundant parallel execution (see Sections 2.2 and 8.3 for a detailed discussion). For example, checkpointing can gracefully deal with increasing load, while redundant execution always requires at least half the processing cycles to be available and devoted to it.

Our solution involves the following subproblems.

Query Partitioning. Each server needs to partition its query graph into several subgraphs so that each subgraph can be assigned a different backup server. We refer to these subgraphs as *HA units*. We study the problem of forming HA units as well as preserving safety against failures while the system reforms HA units.

Backup Assignment. We need an algorithm that finds an appropriate backup server for each HA unit. Our backup assignment algorithm balances the checkpoint load and minimizes the expected recovery time.

Checkpoint Scheduling. As we rely on checkpointing for HA, we need a method that determines the order and the frequency of checkpoint for HA units. Our scheduling algorithm takes into account the characteristics of HA units such as processing load and checkpoint cost.

In summary, our cooperative, fine-grained HA approach has the following advantages.

1. Since the recovery load is distributed and balanced over multiple servers, the approach facilitates faster recovery than previous ones.
2. Each server checkpoints only a small fraction of its query graph at each step. Therefore, the additional latency incurred by HA tasks is kept significantly lower than that of previous ones.
3. Each server strives to fully use its spare CPU cycles (i.e., those left after regular processing) to optimize the recovery performance.
4. If a server fails, each backup server takes over only a fragment of the query graph from the failed server. For this reason, after recovery, each server is likely to experience only a small increase in its processing load, thus keeping its latency under control.
5. The framework is adaptive and does not require human

*This material is based upon work supported by the National Science Foundation under Grants No. IIS-0325838 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.

administration (e.g., no primary/backup designation).

In addition to the HA solution, we dissect the execution details of stream operators and develop low-cost checkpoint mechanisms. We also construct analytic models for both checkpoint cost and expected recovery time. Finally, we present prototype- and simulation-based results that substantiate the utility of our work.

The rest of this paper is organized as follows. We provide an overview of the topic in Section 2 and devise our backup framework in Section 3. We discuss HA unit formation in Section 4 and present algorithms for checkpoint scheduling and backup assignment in Sections 5 and 6, respectively. In Section 7, we analyze stream operators and design efficient delta-checkpointing techniques. We demonstrate the experimental results in Section 8 and cover the related work in Section 9. We conclude in Section 10.

2 Background

In stream processing, a query is expressed in the form of a directed acyclic graph of operators that define how to transform the stream data [6, 4, 7]. Some stream operators are directly borrowed from relational algebra (e.g., filter, map, union) and others are adapted to operate over continuous data streams (e.g., aggregate, join) [3]. The latter execute based on their bounded views, called *windows*, over the data streams to cope with the infinite nature of streams.

In a DSPS, the stream operators are distributed over multiple servers in a scalable manner [9, 20, 24]. We call the mapping between the operators and the servers that execute them a *query deployment plan*. Formally, given a set of servers $\{S_i\}_{i=1}^{n}$ and a query network Q (i.e., the union of all queries submitted), we denote a query deployment plan with $\{Q_i\}_{i=1}^{n}$, where each Q_i represents the set of operators that server S_i runs. We call Q_i the query on server S_i.

2.1 Previous HA Solutions for Stream Processing

Recently, a few HA techniques have been proposed for stream processing [19, 13, 5]. In those techniques, some prechosen servers (called *primary* servers) run queries and other servers act as *backup* servers. Each backup server is responsible for detecting the failure of its primary and immediately taking over the primary's execution. These approaches provide significantly faster recovery than traditional log-based "restart recovery" techniques [16].

Stream-oriented HA solutions primarily differ in how the backup servers operate. Our previous paper [13] proposed the following three alternatives. In *passive standby*, each primary periodically copies to its backup *only the changed part* in its state. We call this task *delta-checkpoint* (or shortly *checkpoint*). Unlike passive standby, *active standby* uses *redundant execution*, in which each backup also receives the input data (from upstream servers) and processes them in parallel with its primary. Finally, in *upstream backup*, each primary logs its output data while the backup remains inactive. If a primary fails, the backup rebuilds the primary's state from scratch by processing tuples logged at upstream servers. Approaches in [19, 5] fall into the active standby model.

2.2 Choosing Checkpoint as the HA method

Each HA method mentioned in Section 2.1 has unique relative benefits in terms of network utilization, recovery speed, recovery semantics, and the effect on regular processing [13]. In this paper, we choose to focus on *checkpointing* (i.e., passive standby) as the underlying HA method. As we argue below, our choice is primarily due to the observation that checkpointing can be used to effectively address a larger set of workload and configurations than other alternatives.

First, we do not use upstream backup as it would not effectively support operators with large windows (e.g., recovering an operator with a window size of 10 minutes requires re-processing 10 minutes worth of tuples). Second, we do not consider active standby because it may not withstand the *high load* situations that checkpointing can tolerate. More specifically, active standby requires the backup to consume the same amount of resources as the primary. Otherwise, the backup will fall behind the primary and eventually will fail to provide fast recovery. By contrast, checkpointing incurs significantly less overhead as it copies only the *remaining result* of the processing since the last checkpoint (e.g., for an aggregate operator, it needs to copy only the most recent summary value rather than all the previous values). In Sections 8.2 and 8.3, we experimentally demonstrate this point. Checkpointing also has an advantage that it can easily handle *non-deterministic* operators, whereas the other techniques require complex solutions [13].

3 Our Backup Model

In contrast to the previous HA models, our model enables parallel recovery over multiple machines. We begin this section by stating the assumptions behind this model. Then, we describe the overall framework as well as the operation during both non-failure and failure periods.

3.1 Assumptions

System Configuration. We assume that servers are grouped into *clusters* (e.g., those of 5-20 servers) and study how the servers in each cluster can cooperate to achieve HA.

Communication. We assume that servers in the same cluster are connected with a fast, reliable network (e.g., gigabit LAN). The communication protocol guarantees robust message delivery and also preserves message ordering. We do not consider network failures that isolate server clusters [5].

Failure Model. We assume that all servers are subject to failure and a failed server stops functioning (i.e., fail-stop). We also assume that a server failure is a rare event and thus aim at protection against *single* server failures. It is generally acknowledged that a *1-safety* guarantee is sufficient for most real-world applications [11].

Processing Load. We assume that the overall processing load is most of the time under the system's processing capability and well balanced over the servers (for this reason, each server can initiate HA tasks at idle times). We do not consider medium- to long-term overload situations because they in general necessitate load shedding [22] to favor timeliness over correctness, contradicting the principle of HA.

3.2 The Basic Architecture

In this subsection, we describe the overall organization of the HA framework. We assume checkpoint as the means to maintain backups (refer to Section 2.2 for the reasons) and use Figure 1 for illustration. Once the system launches the query network, the query on each server can be viewed as

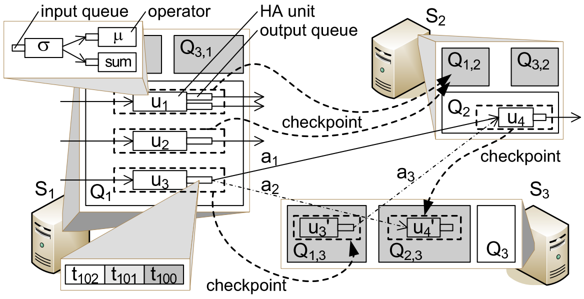


Figure 1. The Backup Model

a set of connected subgraphs. For example, on server S_1, a filter (σ), a map (μ) and an aggregate (sum) form a subgraph u_1. We take such a *maximal connected subgraph* as an *atomic unit* for checkpoint because checkpointing only part of it would yield an inconsistent backup image. Notice that, for a chain where operator o_1 outputs to o_2, checkpointing only o_1 will leave (on the backup server) an image that would result from an invalid execution where some recent output of o_1 disappears without being fed into o_2. For this reason, we call such a subgraph an *HA unit*. Hereafter, we specify the query deployment plan in terms of HA units. In Figure 1, $Q_1 = \{u_1, u_2, u_3\}$, $Q_2 = \{u_4\}$, and so forth. Operators that belong to an HA unit are called *constituent operators*.

We also regard HA units on the same server as *independent* units for checkpoint (i.e., can be checkpointed onto *different servers* at *different times*) because they have no interdependency with each other. A *backup assignment*, a mapping between HA units and their backup servers, is denoted by $\{Q_{i,j} : i \neq j, 1 \leq i, j \leq n\}$, where $Q_{i,j}$ is the set of HA units that server S_i executes and server S_j backs up ($i \neq j$ implies that a server cannot back up itself). We call each $Q_{i,j}$ an *HA segment*. In Figure 1, $Q_{1,2} = \{u_1, u_2\}$, $Q_{1,3} = \{u_3\}$, and $Q_{2,3} = \{u_4\}$. u_3' and u_4' are the backup images built by checkpointing u_3 and u_4 onto S_3, respectively. Each shaded area $Q_{i,j}$ represents the collection of backup images that S_j maintains for S_i.

It should be noted that the formation of HA units will significantly affect the behavior of our HA framework. We discuss this issue in Section 4.

3.3 No Loss Guarantee

Our HA model masks a server failure by making other servers in the cluster collectively rebuild the latest state of the failed server. For this model, we assume that a backup can precisely repeat the pre-failure execution of its primary as long as it can obtain the input tuples that the primary has processed since the last checkpoint. We refer the reader to our previous work in [13] for the details of providing this guarantee for all kinds of stream-processing operators including nondeterministic ones.

In Figure 1, if S_2 fails, the processing that involves u_4 no longer continues. In this case, S_3 has to take over the processing because it is the only one that owns the backup image u_4' of u_4. In more detail, S_3 has to set up a new connection a_2 to feed its backup image u_4' and start executing u_4' to recover the state of u_4. To prevent data loss in this case, each backup image must be able to obtain the tuples that the primary has processed since the last checkpoint. In other words, if S_2 processed tuple t_{100} after checkpointing u_4 onto S_3, u_4' on S_3 must be able to receive that tuple through a_2. For this reason, each HA unit has *output queues*, one for each output, to retain such tuples (i.e., those that the downstream backups are currently missing). Those tuples can be safely discarded when the downstream server processes them and checkpoints the effect onto the backup server. In our current implementation, both output queues and backup images are built in memory. This implementation can be extended to spill those components to disk under memory contention.

3.4 Non-Failure Time Operation for HA

As stated in Section 3.1, we assume that each server has spare CPU cycles and uses them for HA purposes. An idle server (i.e., one that has no tuples to process at the moment) can perform one of the following HA tasks:

- **Capture:** The server chooses an HA unit and sends to the backup server a message that captures the delta in the state (i.e., what has changed since the last checkpoint). We use the terms *capture* and *checkpoint message*, respectively, to refer to the task and the message that the task constructs.
- **Paste:** The server chooses one of the checkpoint messages that it received, and copies the content of the message to the corresponding backup image. We call this task *paste*. Once a paste finishes, the server notifies the sender of the checkpoint message (i.e., the primary server for the checkpointed HA unit). This enables the next round of checkpoint for that unit.

In the rest of this section, we describe HA tasks in detail. We discuss the problem of scheduling them in Section 5.

Contents of a Checkpoint Message. When a server begins capturing an HA unit, the input queues of the HA unit (i.e., those of the constituent operators) are empty. For this reason, the server skips input queues and captures only the *constituent operators* (discussed in Section 7) and *output queues*. Notice that only a small part of each output queue needs to be captured. In Figure 1, if S_2 acknowledged to S_1 that it received t_{101}, S_1 needs to capture only t_{102} among the tuples in the output queue of u_3. With t_{102}, the output queue of u_3' on S_3 can guarantee that a_3, a stream that will flow if S_1 fails, will not miss any tuple.

Checkpoint vs. Processing. Once a capture task begins, the server defers stream processing (i.e., only buffers arriving input tuples) until the capture ends. This is because (1) executing an HA unit while it is being captured might introduce inconsistency in the captured image (i.e., capture and processing conflict semantically) and (2) interrupting a capture to execute other HA units will further suspend the HA unit currently being captured. If an exceptional input burst appears during a capture, it may be desirable to abort the ongoing capture and immediately resume the processing to bound the growth of the latency. However, such an abortion is not always useful because the change in state tends to grow over time (i.e., a later capture is usually more expensive). In contrast to capture, paste can be interrupted to execute other HA units. Notice that this task on backup can be done in parallel with the execution of the HA unit on the primary.

Once a capture finishes, a *processing burst* appears until the buffered input tuples are consumed. Conceptually, the duration of capture (i.e., capture cost) implies the penalty of HA (i.e., the additional processing latency due to capture). Furthermore, both the capture cost and the server's processing load affect the duration of the processing burst and also the checkpoint interval. In practice, we can set a lower bound on the checkpoint interval to prevent checkpointing too frequently (i.e., to trade off processing against HA). We can also set an upper bound that forces a checkpoint to bound the growth of recovery time (i.e., to trade off HA against processing).

3.5 Failure and Recovery

In our HA model, each server S is monitored by a designated server that periodically (e.g., every 100ms) pings S. If S does not respond for a timeout period (e.g., 300ms), the server assumes that S has failed and broadcasts this to the other servers in the cluster. Each of these notified servers then searches for checkpoint messages from the failed server and pastes those messages to the corresponding backup images. Next, it finds the backup images that it has maintained for the failed server and begins executing them as its new HA units, while redirecting the input and output streams as described in Section 3.3. Finally, the server lets those backed-up HA units catch up with other HA units.

We use the term *recovery* to refer to the process during which the alive servers in the cluster take the actions described above. When the servers collectively rebuild the latest state of the failed server, we say that the cluster has *recovered* from the failure. Note that *having recovered* does not necessarily imply *being able to mask the next failure*. This is because the system may not be able to tolerate the next failure until it secures, again by use of checkpoint onto new backup servers, the HA units taken over during recovery. The *period of instability* refers to the amount of time, after failure, until all HA units are again protected. Finally, if the failed server comes back up, it joins the system as a new member.

4 Formation of HA Units

As illustrated in Section 3.2, query deployment determines the formation of HA units. We start this section by showing that the load management principles for stream processing are also beneficial to our HA framework. We then introduce a strategy that avoids managing too many HA units. Finally, we discuss preserving the safety guarantee while the system reforms HA units.

4.1 Impact of Load Management Principles

One principle of load management in stream processing is to distribute operators with highly correlated loads over different machines [24]. This is because placing them on the same machine will make it more vulnerable to load spikes. Adjacent operators (i.e., those connected by data streams) usually exhibit high load correlation. Therefore, they (except those with very low processing load) tend to be placed at different servers. Furthermore, operators with heavy processing load are usually split into smaller pieces and distributed over multiple servers [9, 20] due to their negative impact on load management (refer to [23] for quantitative analysis). For the two reasons above, each server is likely to own many small-size operator chains (i.e., many fine-grained HA units). This is indeed advantageous to our HA model because (1) more HA units, in general, lead to better backup distribution (see Section 1 for the benefits) and (2) finer HA units tend to have smaller capture costs (i.e., smaller disruption in processing). Note that HA units with high capture costs can also be split in order to lower the costs.

4.2 Merging HA Units

While having many HA units is usually beneficial in terms of backup distribution, managing too many HA units may incur significant overhead. We address this problem by iteratively *merging* HA units with similar characteristics as long as the capture cost remains under a threshold. In other words, we put all the operators that constitute those HA units into a new HA unit, even though those operators do not form a connected graph. We use the ratio of *processing load* over *capture cost* as the similarity metric because this is what our checkpoint scheduler uses to optimize the recovery speed (refer to Section 5). Merging HA units that are similar in this manner avoids making bad scheduling decisions. Notice that the merged HA units will get checkpointed with a similar frequency as before the merge.

4.3 Safety during HA Unit Reformation

Safety against failure has to be preserved even while the system reforms HA units due to operator splitting and migration. We achieve this by keeping the old backup images of the involved HA units, until the reformation completes and the newly formed HA units are again backed up. Due to the page limitation, we cover only the following representative case. Suppose that an operator ρ is migrated to server S_i and added (as a constituent operator) to an HA unit u on S_i. In this case, the previous backup server S'_j for ρ has to keep its backup image ρ' of ρ until S_i checkpoints the expanded version of u onto the backup server S'_i for u. This is because, before the checkpoint, S'_j is the only one that backs up ρ. S'_j can safely remove its backup image ρ' after S'_i receives the checkpoint message that captures $u \ni \rho$.

5 Checkpoint Scheduling

In our backup model, an idle server can perform either a *capture* task (i.e., among the HA units not being checkpointed, choose one, compose a checkpoint message for that one, and send the message to the right backup server) or a *paste* task (i.e., among the checkpoint messages received, choose one and copy the content of the message to the right backup image). In this section, we devise an algorithm that schedules such tasks in a manner that minimizes the expected recovery time. We first discuss how we can find the capture task that will most reduce the expected recovery time. Then, we describe how we choose the best from both capture and paste tasks. We end this section discussing the key properties of our scheduling algorithm.

5.1 Choosing the Best Capture Task

A capture task sends the changed part of an HA unit's state to the backup server. The backup server then can freshen the backup image (i.e., reduce the amount of work to do during recovery) by simply copying the delta received. However, as we describe below, the recovery time is heavily dependent on how a server schedules capture tasks.

Figure 2 illustrates an example where server S_1 check-

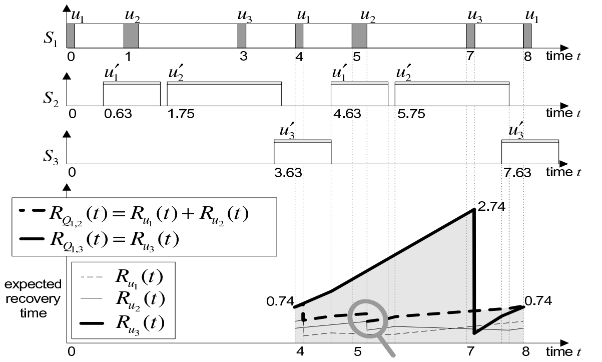


Figure 2. Reovery Time (Round-Robin)

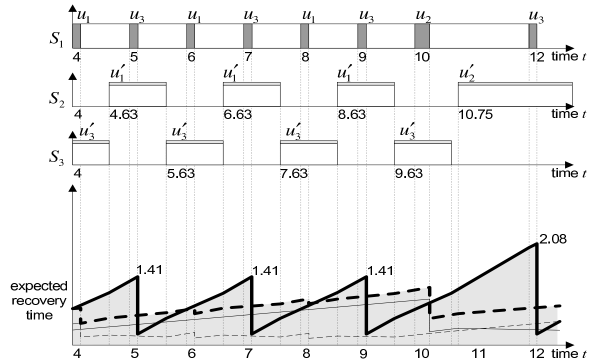


Figure 3. Reovery Time (Min-Max)

points its HA units u_1 and u_2 onto S_2, and u_3 onto S_3 in a round-robin fashion. To ease illustration, we do not consider the paste tasks that S_1 would perform and the capture tasks that S_2 and S_3 would do. We also ignore network latency and assume that HA units on S_1 have constant processing loads (in terms of CPU utilization): $l_{u_1}(t) = 11\%$, $l_{u_2}(t) = 10\%$, and $l_{u_3}(t) = 66.5\%$ for all time t. Finally, we assume that those units have (1) constant capture costs (in seconds): $c_{u_1}(t) = 0.125$, $c_{u_2}(t) = 0.25$, and $c_{u_3}(t) = 0.125$ for all time t and (2) the same paste costs as capture costs: $c'_{u_k}(t) = c_{u_k}(t)$ for $k = 1, 2, 3$. Notice that these costs are the amounts of time that the CPU would consume when it performs HA tasks in isolation. The assumptions above are to ease illustration (refer to Sections 8.1 and 8.2 for the details in real cases).

As described in Section 3.4, each capture (represented as a dark rectangle in Figure 2) defers query processing. For this reason, a processing burst (represented as an empty rectangle) appears after that. As mentioned before, the duration of such a burst, is a function of the capture cost and the server's processing load. For example, the duration of the burst after capturing u_1 is $\frac{\texttt{[capture cost][total load]}}{1-\texttt{[total load]}} = \frac{c_{u1}(t)\left(l_{u_1}(t)+l_{u_2}(t)+l_{u_3}(t)\right)}{1-\left(l_{u_1}(t)+l_{u_2}(t)+l_{u_3}(t)\right)} = \frac{0.125\cdot 0.875}{1-0.875} = 0.875$ (seconds). The figure also illustrates that each paste is deferred until its turn and, in contrast to captures, is interleaved with query processing (refer to Section 3.4 and the stacks of grey and empty rectangles in the figure).

Figure 2 also demonstrates how the expected recovery time changes over time for various entities such as HA units u_1, u_2, u_3, HA segments $Q_{1,2}$, $Q_{1,3}$ and the query Q_1 on server S_1. We use the convention that $R_*(t)$ represents the expected amount of time to recover an entity $*$ if the primary for $*$ fails at time t (due to the page limitation, we leave the formal definitions in [14]). The figure shows that capturing u_2 during $[5, 5.25]$ reduces $R_{u_2}(t)$ from 0.43 to 0.28 when it finishes at 5.25. Notice that when the capture is about to end, the expected amount of time to recover u_2 is $\int_1^{5.25} l_{u_2}(\tau)d\tau = 0.43$ because the backup image u'_2 on S_2 has the state of u_2 as of time 1. However, when the capture finishes at time 5.25, S_2 receives a checkpoint message that captures u_2 as of time 5. Therefore, if S_1 fails, S_2 will take $c'_{u_2}(t) = 0.25$ seconds to consume the checkpoint message and $\int_5^{5.25} l_{u_2}(\tau)d\tau \simeq 0.03$ seconds to replay the execution of u_2 that occurred from time 5 to 5.25 on S_1.

The capture task for u_2 also reduces $R_{Q_{1,2}}(t)$ by the same amount. This is because $R_{Q_{1,2}}(t) = R_{u_1}(t) + R_{u_2}(t)$ in the example, as u_1 and u_2 on S_1 are backed up at S_2. By contrast, the capture task *cannot reduce* $R_{Q_1}(t)$ (see the upper bound of the grey area). Notice that the system will recover from S_1's failure only if both S_2 and S_3 recover segments $Q_{1,2}$ and $Q_{1,3}$, respectively. Formally, $R_{Q_1}(t) = \max\{R_{Q_{1,2}}(t), R_{Q_{1,3}}(t)\}$. However, capturing $u_2 \in Q_{1,2}$ does not relieve the largest recovery load on S_3 (see in the figure that $R_{Q_{1,3}}(t) > R_{Q_{1,2}}(t)$). *To reduce $R_{Q_1}(t)$, S_1 at time 5 should have started capturing $u_3 \in Q_{1,3}$* (as in Figure 3) *rather than $u_2 \in Q_{1,2}$.*

Based on this observation, we design an algorithm that selects an HA task that *will minimize the maximum recovery load* among those spread over other servers. For this reason, we call our scheduling algorithm "*min-max*". In detail (also refer to Figure 4), each server S_i looks for all HA units $u \in Q_{i,j}$ such that (1) $u \in \texttt{c-q}$ and (2) $R_{Q_{i,j}}(t + c_u(t)) = R_{Q_i}(t + c_u(t))$, where $\texttt{c-q}$ is a queue that remembers the HA units that S_i can checkpoint immediately (i.e., those not being checkpointed) and $t + c_u(t)$ is the time when capturing u will end. Condition (2) above implies that u's backup server will have *the maximum recovery load when capturing u is about to finish.* Among such HA units, the server finds HA unit u^* such that capturing it will *most reduce the recovery time*, relative to the capture cost. As the metric for this, the server uses $\frac{\Delta R_u(t)}{c_u(t)}$ for each HA unit u, where $\Delta R_u(t)$ denotes the reduction in recovery time at the cost $c_u(t)$ of capturing u. We define $\Delta R_u(t)$ as $\int_{\alpha_u(t)}^t l_u(\tau)d\tau - c'_u(t)$, where $\alpha_u(t)$ denotes the start time of the previous capture. In the definition of $\Delta R_u(t)$, the first and second terms represent the *gain* of freshening the backup image and the *penalty* of consuming the checkpoint message, respectively. Notice that our algorithm is prefers HA units with *high processing load* (see $l_u(\tau)$ in the numerator of the metric) and *low checkpointing cost* (see $c_u(t)$ and $c'_u(t)$ in the metric).

5.2 Capture vs Paste

In principle, a server conducts *capture tasks* to better prepare for the failure of itself and *paste tasks* for the failure of others. To strike a balance between these goals, each server finds the HA task, no matter it is a capture or a paste, that will assist the HA segment with the largest recovery load. In detail, server S_i first computes $R_{Q_{i,j^*}}(u^*, t)$, the expected recovery time for the moment when it finishes capturing the

Whenever S_i forms an HA unit $u \in Q_{i,j}$
01. $\texttt{c-q}$.push(u, i); // enqueue the HA unit
Whenever Idle
02. $\mathcal{C} \leftarrow \{(u, j) \in \texttt{c-q} : R_{Q_{i,j}}(t + c_u(t)) = R_{Q_i}(t + c_u(t))\}$
03. find $(u^*, j^*) \in \mathcal{C}$ such that // find the best HA unit to capture
$\frac{\Delta R_{u^*}(t)}{c_{u^*}(t)} = \max\{\frac{\Delta R_u(t)}{c_u(t)} : (u, j) \in \mathcal{C}\}$
04. for each j $(1 \leq j \leq n,\ j \neq i)$
05. $(u, \Delta) \leftarrow \texttt{p-q}[j]$.first() // oldest checkpoint msg from S_j
06. if $\left(R_{Q_{j,i}}(u, t) > R_{Q_{i,j^*}}(u^*, t)\right)$
07. if $\left(u' = \texttt{null} \mid\mid R_{Q_{j,i}}(u, t) > R_{Q_{j',i}}(u', t)\right)$
// if no best paste so far or if the current one is the best
08. $(u', j', \Delta') \leftarrow (u, j, \Delta)$ // remember the best paste
09. if $(u' = \texttt{null})$ // if no paste is better than the best capture
10. capture(u^*, j^*); // do the best capture
11. remove (u^*, j^*) from $\texttt{c-q}$;
12. else // if there exists a paste better than the best capture
13. paste(u', j', Δ') // do the best paste
14. remove (u', Δ') from $\texttt{p-q}[j']$;
capture(u^*, j^*)
15. copies into Δ the recent change in u^*;
16. S_{j^*}.$\texttt{p-q}[i]$.push(u^*, Δ); // send Δ to the backup
paste(u', j', Δ')
17. copies Δ' into u';
18. $S_{j'}$.$\texttt{c-q}$.push(u, i); // notify the primary

Figure 4. Min-Max Scheduling (on server S_i)

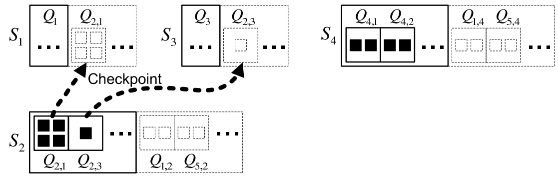
best HA unit $u^* \in Q_{i,j^*}$ (we formally define this in [14]). Then, for each backup segment $Q_{j,i}$, it computes the expected recovery time for the moment when it completely consumes the *oldest pending* checkpoint message from S_j (this FIFO order is to obey the decisions that S_j made). Using $u \in Q_{j,i}$ to denote the HA unit captured in the checkpoint message, we represent such expected recovery time as $R_{Q_{j,i}}(u, t)$ (again see [14] for the formal definition). If $R_{Q_{j,i}}(u, t) > R_{Q_{i,j^*}}(u^*, t)$, we assume that backup segment $Q_{j,i}$ needs more care than primary segment Q_{i,j^*} (i.e., the paste for $u \in Q_{j,i}$ is more urgent). The server selects the best from capture and paste tasks based on this rationale.

5.3 The Complete Scheduling Algorithm

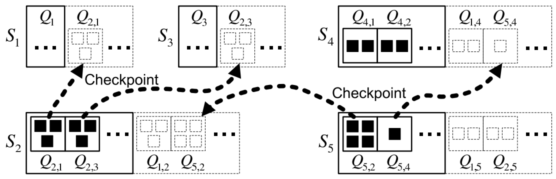
Figure 4 summarizes the min-max algorithm. Whenever a server forms a new HA unit, it pushes it into $\texttt{c-q}$ (line 01). An idle server first finds the best capture task (lines 02-03). Then, it attempts to find the paste task that is more effective than all others (including the best capture task) (lines 04-08). Finally, it performs the best task found. If a capture task is chosen (lines 09-11), the server composes a checkpoint message and sends it to the relevant backup server (lines 15-16). If a paste task is chosen (lines 12-14), the server consumes the checkpoint message and then notifies the completion of checkpoint to the primary (lines 17-18).

5.4 Discussion

Our min-max algorithm selects the HA task that will most reduce the largest recovery load. Figures 2 and 3 show an example where our scheduling algorithm maintains the recovery time at a 30% lower level than round-robin (in Section 8, we show the results from real test cases). The figures also show that min-max takes a longer time than round-robin until it checkpoints each HA unit at least once (we call such a period of time a *checkpoint cycle*). This is because min-max frequently checkpoints HA units with high processing load and low checkpoint cost, yielding a non-uniform schedule. Figure 3 shows that the recovery time under min-max gradually increases until it eventually drops at the end of a checkpoint cycle. This is because the backup images of uncheckpointed HA units get staler over time.



(a) Unbalanced Backup Assignment



(b) Too Many Backup Images

Figure 5. Backup Reassignment

6 Dynamic Backup Assignment

In our HA model, the recovery time depends on the checkpoint schedule but also on the backup assignment. For example, a server with too many backups will easily be the bottleneck that slows down the circulation of HA tasks. If this happens, other servers will poorly prepare for failure as they cannot efficiently use spare CPU cycles. In this section, we study how we can avoid this problem. We assume that servers with low backup load volunteer to back up new HA units whenever they appear. Hereafter, we focus on modifying the backup assignment to cope with varying system conditions introduced by operator migration, changes in input rates, etc.

6.1 Determining Backup Load Imbalance

In principle, our algorithm reassigns backups to assist the server whose failure will cause the longest recovery. We call such a server *the worst point of failure*. We take this approach because a server usually becomes the worst point of failure due to unbalanced backup distribution and also because it is always advantageous to improve the worst-case disruption that a failure would cause. We use Figure 5 to illustrate two typical cases of backup imbalance. We assume that S_2 is the worst point of failure and, to ease presentation, all the HA units (i.e., those represented as dark small rectangles) have identical properties. Backup images are represented as small dotted-line rectangles. Figure 5(a) shows the case where S_2 has poorly assigned backups (see that S_1 backs up too much for S_2). In this case, S_2 can resolve the problem by transferring part of S_1's backup responsibility to S_3. Figure 5(b) shows another case where S_2 keeps too many backup images for others (see $Q_{1,2}$ and $Q_{5,2}$). In this case, S_2 should not reassign backup servers as it cannot improve any further. Instead, a different server (say S_5) should do the task for S_2.

6.2 The Backup Reassignment Algorithm

Figure 6 summarizes our backup reassignment algorithm. At the end of *epoch E* (the period required for every server to

On server $S_{\overline{i}}$ with the highest recovery time, whenever epoch E ends

01. Find $\underline{k}, \overline{j}, \underline{j}$ such that
$R_{Q_{\overline{i},\overline{j}}}(E) = \max_{j=1}^{n} R_{Q_{\overline{i},j}}(E)$
$R_{Q_{\overline{i},\underline{j}}}(E) = \min_{j=1}^{n} R_{Q_{\overline{i},j}}(E)$
$R_{Q_{\underline{k}}}(E) = \min_{k=1}^{n} R_{Q_{k}}(E)$
02. if $R_{Q_{\overline{i},\underline{j}}}(E) < R_{Q_{\underline{k}}}(E)$ // if the backup load is unbalanced
03. move($\overline{j}, \underline{j}$) // balance the backup load
04. else // if $S_{\overline{i}}$ has too many backup images
05. find $\overline{k} = \arg\max_{k=1}^{n} \left[R_{Q_{k,\overline{i}}}(E) - R_{Q_{k,\underline{k}}}(E)\right]$
06. $S_{\overline{k}}$.move($\overline{i}, \underline{k}$) // let $S_{\overline{k}}$ balance the backup load

$S_{\overline{i}}$**.move(**$\overline{j}, \underline{j}$**)**

07. $\Delta h \leftarrow \frac{R_{\overline{i},\overline{j}}(E) - R_{\overline{i},\underline{j}}(E)}{R_{\overline{i},\overline{j}}(E)} h_{Q_{\overline{i},\overline{j}}}(E)$ // backup load to transfer
08. for each $u \in Q_{\overline{i},\overline{j}}$
09. if $h_u(E) \leq \Delta h$ // if finds an HA unit to reassign the backup
10. $\Delta h \leftarrow \Delta h - h_u(E)$ // update the backup load to move
11. $Q_{\overline{i},\overline{j}} \leftarrow Q_{\overline{i},\overline{j}} - \{u\}$ // reassign the backup server
12. $Q_{\overline{i},\underline{j}} \leftarrow Q_{\overline{i},\underline{j}} \cup \{u\}$

Figure 6. Backup Reassignment

finish at least one checkpoint cycle), the servers in the cluster determine the worst point of failure $S_{\overline{i}}$, based on the expected recovery times averaged over the epoch. We use this periodic approach because (1) it is hard to know the recovery time in an average sense before a checkpoint cycle ends (see Section 5.4) and (2) we should avoid changing backup assignment too frequently. Note that checkpointing an HA unit onto a new backup server (i.e., a whole checkpoint) is usually more expensive than an ordinary delta-checkpoint.

Server $S_{\overline{i}}$ (the worst point of failure) first finds its HA segments $Q_{\overline{i},\overline{j}}$ with the largest recovery load and $Q_{\overline{i},\underline{j}}$ with the smallest recovery load (note that $R_{Q_{\overline{i},\overline{j}}}(E) > R_{Q_{\overline{i},\underline{j}}}(E)$ by definition). It also finds another server $S_{\underline{k}}$ whose failure will result in the shortest recovery (line 01). If $R_{Q_{\overline{i},\underline{j}}}(E) < R_{Q_{\underline{k}}}(E)$, $S_{\overline{i}}$ assumes that $S_{\underline{j}}$ is assigned too low backup load and accordingly balances the backup load between $S_{\overline{j}}$ and $S_{\underline{j}}$ (lines 02-03; in Figure 5(a), $S_{\overline{j}}$ and $S_{\underline{j}}$ correspond to S_1 and S_3, respectively). Otherwise (line 04), it assumes that it has too many backup images. Thus, it locates a different server $S_{\overline{k}}$ that will effectively balance the backup load between $S_{\overline{i}}$ and $S_{\underline{k}}$ (lines 05-06). In Figure 5(b), $S_{\overline{k}}$ corresponds to S_5. $S_{\overline{i}}$ and $S_{\underline{k}}$ correspond to S_2 and S_4, respectively.

The **move(**$\overline{j}, \underline{j}$**)** phase in Figure 6 shows the details of reassigning backup servers. In the algorithm, $h_u(E)$ denotes the total CPU cycles used for updating the backup image of HA unit u during epoch E (we call this quantity the *backup load* of HA unit u). Similarly, $h_{Q_{\overline{i},j}}(E)$ represents the backup load of HA segment $Q_{\overline{i},j}$. To balance the backup load, $S_{\overline{i}}$ first computes the amount of backup load Δh to move from segment $Q_{\overline{i},\overline{j}}$ to segment $Q_{\overline{i},\underline{j}}$ (line 07). Then, it reassigns backup servers until the amount of backup load transferred reaches Δh (08-12). Note that this **move** phase only chooses the new locations to put backup images. The first checkpoints after reassignment in fact create the backup images.

7 Delta Checkpointing

An efficient checkpointing mechanism will shorten the duration of HA tasks. This implies better runtime performance (because the disruption in processing will decrease) and faster recovery speed (because more frequent checkpoints will be possible). In this section, we describe how to implement efficient, operator-specific delta-checkpointing techniques based on the details of operators. We also construct the associated cost model. In Section 8, we demonstrate that checkpoint costs can be estimated accurately relying on the cost model. We also show that our min-max algorithm performs well based on such cost estimations.

As described in Section 3.4, capturing an HA unit requires incorporating the states of constituent operators and, for each output queue, a round trip worth of tuples. The latter however can be ignored safely by holding the checkpoint message until the downstream servers acknowledge the receipt of the tuples. Thus, we can represent the cost $c_u(\alpha)$ of capturing an HA unit u as $\sum_{\rho \in u} c_\rho(\alpha)$, where α is the start time of the capture task and $c_\rho(\alpha)$ is the cost of capturing the internal data structure of a constituent operator ρ. Because stateless operators will not incur any checkpoint cost, we design (and analyze) the delta-checkpointing methods for two representative stateful operators, aggregate and join.

7.1 Aggregate

Aggregate splits input stream I into substreams $\{I[g]\}$, one for each group-by value g. For each substream $I[g]$, it assumes windows (sets of tuples) of w seconds that appear every s seconds (we can also define windows in terms of tuple counts). Whenever a window expires, the operator outputs an aggregated value computed from the tuples contained in the window. In more detail, for each input tuple, the operator (1) reads (from the tuple) timestamp t and group-by value g. Next, the operator (2) uses its "map" to quickly locate *the list of windows* associated with g (if none exists, it creates a new list), and (3) determines if it needs to add new windows (e.g, if the timestamp t of the tuple is 401.5 and the timestamp of the most recent window is 400, an aggregate with step size s of 1 second will add a window anchored at time 401). Then, the operator (4) iterates over the windows in the list updating the summaries (e.g., counts, sums, or histograms, etc.) that those windows maintain. Finally, it (5) closes expired windows while sending their summaries as output tuples.

Delta-Checkpointing. We use one dirty bit for each group-by value to mark those that have appeared since the last checkpoint (we clear dirty bits at the end of capture). We also use one dirty bit for each window to indicate whether it was created after the last checkpoint or not. To capture an aggregate, the primary server finds all the windows associated with group-by values with their dirty bits on. Next, it copies into the checkpoint message (1) the entire content of each window with its dirty bit on (*full capture* for new windows) and (2) only the summary of each window with its dirty bit off (*partial capture* for updated windows). When the backup server consumes the checkpoint message, it checks the captured window images in the message. For a fully-captured image, it creates a new window from the image and associates the window with the right group-by value (i.e., *full paste*). Otherwise, it copies the partial image onto the corresponding window that already exists in the operator (i.e., *partial paste*).

Cost Model. We can represent the cost of capturing this operator as $C_n c_f + C_u c_p$, where C_n is the number of windows created after the last checkpoint, C_u is the number of updated windows, and c_f and c_p are the costs of fully and partially capturing a window, respectively. We can similarly define the paste cost using per-window full/partial paste costs.

7.2 Join

Join takes input streams I_1 and I_2. This operator searches for all pairs of input tuples (one from each input stream) that (1) belong to the same window of size w and (2) match the predicate defined for the operator. Whenever the operator finds such a pair of *matching input tuples*, it produces the concatenation of them as an output tuple.

Delta-Checkpointing. Since the recent change in state is the tuples newly entered the window, each checkpoint captures those tuples. It also captures the upper bound of the tuples that have left the window so that the backup server can remove those tuples from the backup image.

Cost Model. The number of tuples that have entered the window since the last checkpoint can be represented as $(\lambda_1 + \lambda_2) \min(t - \alpha, w)$, where t is the current time, α is the start time of the last checkpoint, and λ_1 and λ_2 are the input rates (i.e., the number of input tuples per second) of input streams I_1 and I_2, respectively. Therefore, the capture cost can be expressed as $(\lambda_1 c_1 + \lambda_2 c_2) \min(t - \alpha, w)$, where c_1 and c_2 are the cost of capturing a tuple from input streams I_1 and I_2, respectively. We can similarly define the paste cost.

8 Experimental Results

In this section, we describe experimental results from both our Borealis DSPS [5, 2, 24, 13] and a detailed simulator. First, we describe how we set up the experiments (Section 8.1). Then, based on the results from Borealis, we investigate how the cost of checkpointing varies depending on the frequency of checkpoints (Section 8.2). We also demonstrate how our technique effectively reduces the recovery time while being minimally intrusive to regular query processing (Section 8.3). Finally, we present supplementary results obtained from the simulator (Section 8.4).

8.1 The Setup

In our experiments, we used a five server cluster where a 1GBps router interconnects servers with an AMD Sempron 2800+ CPU and 1GB main memory. We used two different input streams. The first (Type 1) is a wide-area TCP trace obtained from [1]. We extract the timestamp and the source-IP address from each packet to form an input feed that runs at 2.0K tuples/sec on average. As commonly observed, the trace has a widely varying stream rate (std = 0.7K tuples/sec) and its source-IP addresses have a highly skewed distribution. The second (Type 2) is an artificial input stream with a source-IP-address field that ranges uniformly from 0 to 99, with a constant input rate of 100 tuples/sec. These two input streams were designed to represent dissimilar loads.

Our test query consists of aggregate operators each of which every second counts the number of tuples for each source-IP address over a window of 10 seconds. We form an HA unit for two parallel aggregates fed by a single input stream. On each server, we generate four HA units with input type 1 and another four HA units with input type 2. It should be noted that we use aggregates since they are commonly implemented as described in Section 7.1 and therefore allow us to obtain results with generality. By contrast, join implementations vary drastically, leading to significantly different processing loads for the same input. In Section 8.3, we show how sensitive the recovery time is to processing load.

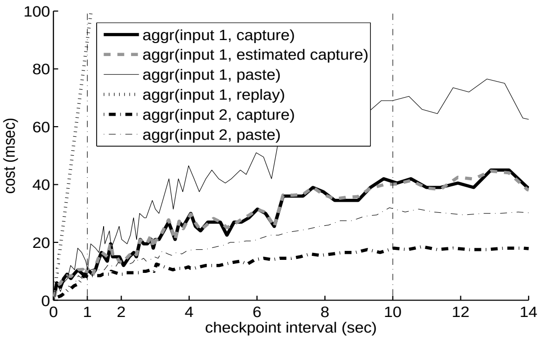


Figure 7. Analysing Checkpointing Costs

8.2 Checkpointing Costs

In this experiment, we vary the frequency of checkpoints and observe how the cost of checkpointing an aggregate operator varies both on primary and backup (Figure 7). As expected, when the checkpoint frequency decreases (i.e., the interval increases), both the time to form a checkpoint (capture cost) and the time to consume a checkpoint (paste cost) increase. This is because the state of the primary will increasingly diverge from the state of the previous checkpoint. Notice that the curves for the type 1 aggregate have jitters, showing how much the checkpoint cost may vary each time due to the burstiness of real packet streams. The figure also shows that type 1 aggregate has a *relatively lower checkpoint cost* than type 2 aggregate, considering its significantly *higher processing load* due to 20 times faster input rate. This is because the former maintains relatively fewer windows as the group-by values have a very skewed distribution.

The curve for aggr(input 1, estimated capture) shows that we can accurately estimate the checkpoint cost. Notice that this is important because our min-max algorithm operates on the basis of cost estimations. In the experiment, we estimated the per-window full/partial capture costs (in μsecs) as c_f=18.6 and c_p=7.3 (see Section 7.1) using linear regression over a collection of triples [# new windows (C_n), # updated windows (C_u), capture cost ($C_n c_f + C_u c_p$)]. Paste costs were estimated with per-window paste costs as c'_f=30.2 and c'_p=7.2.

The figure also shows the bounded nature of operator states: all curves tend to plateau after a 10 second checkpoint interval mainly because the operator can contain at most a 10-second worth of tuples. The curves for the type 2 aggregate flatten out after a 1-second interval, as its input creates at most 100 group-by values. The gradual increase of those curves between 1 and 10 seconds accounts for the increase of new windows created after the previous checkpoint. Note that the type 1 aggregate does not show such flattening (as it continuously observes new IP-addresses) until the checkpoint interval surpasses the window size. The figure also

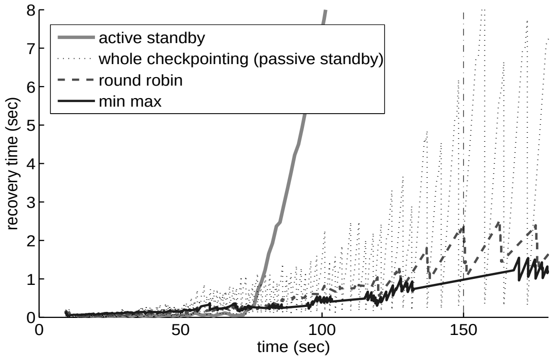


Figure 8. Recovery Time

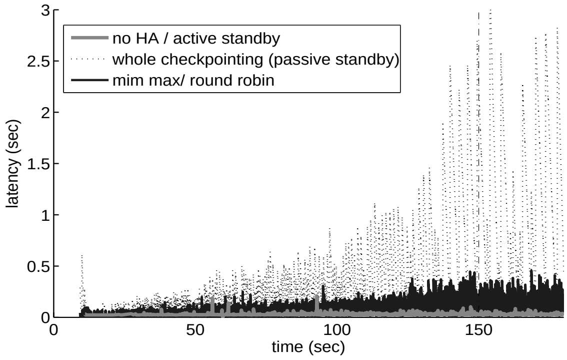


Figure 9. End-to-End Processing Latency

shows that paste costs are usually higher than capture costs as the backup allocates new windows (observe $c'_f > c_f$).

Finally, the difference between (input 1, paste) and (input 1, replay) shows the benefit of checkpointing over active standby. In particular, (input 1, replay) shows how long the backup has to run to make up to date the stale backup image used in the corresponding checkpoint case. Checkpointing uses much fewer CPU cycles due to the reasons in Section 2.2 and the bounded nature of operators as argued above.

8.3 Recovery Time and End-to-End Latency

In this experiment, we study how the *expected recovery time* and *end-to-end latency* change due to HA tasks in our prototype. We increase the stream rate of input type 1 uniformly from 0 (at time 0) to 2.0K tuples/sec (at time 150) while fixing that of input type 2 at 100 tuples/sec. After time 150, input type 1 is fed as described in Section 8.1 and each server is utilized approximately 90% for processing. We deployed a query network of 16 aggregates on each of the five homogenous servers. On each server, we formed one HA unit for the whole checkpointing case and 8 HA units (as described in Section 8.1) for all other cases. The HA units were uniformly assigned backup servers to avoid imbalance in backup load.

Figure 8 shows how the expected recovery time changes as the input rate increases. We can see that active standby cannot withstand as the overall processing load increases beyond 50% of the cluster's computation capability. Observe that backup processes start to fall behind after time 70 as they no longer can use the same amount of resources as primary processes. On the other hand, checkpoint-based methods continue their operations. In general, recovery time is sensitive to the increase in processing load because (1) the recovery load increases accordingly and (2) fewer spare CPU cycles can be devoted to HA tasks. Both round-robin and min-max exhibit significantly faster recovery speed than whole checkpointing, as the recovery load is distributed over multiple servers. Min-max achieves the fastest recovery speed by favoring Type 1 HA units over Type 2 HA units. The jitters in Figure 8 happen when Type 2 HA units eventually get checkpointed near the end of a checkpoint cycle (refer to Section 5.4).

Figure 9 shows how the HA tasks affect query processing. We do not present the curve for active standby as its latency is similar to the latency without any HA method. Notice that, to be fair, we assumed a distributed adaptation of the basic active standby model that can flexibly trade off processing against HA in overload situations. The figure also shows that fine grained checkpoint techniques disrupt regular processing much less than the standard whole checkpointing approach. Each spike in latency is introduced by an HA task. Round-robin behaves similarly to min-max.

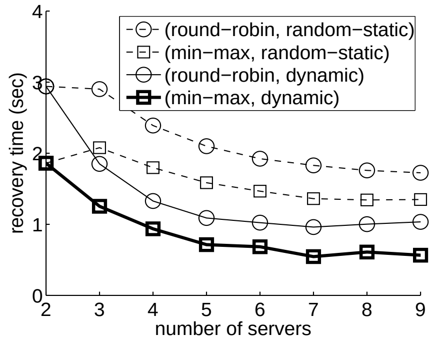


Figure 10. Scheduling & Backup Assignment Effects

8.4 Scheduling and Backup Assignment

Figure 10 shows how recovery time changes as we increase the number of servers for combinations of scheduling algorithms (round-robin and min-max) and backup assignment techniques (random-static and dynamic). This result was obtained from our detailed Borealis simulator. Round-robin scheduling and random-static assignment are considered to be the baseline cases. We compare the more robust algorithms, min-max and dynamic, with the baseline.

The first thing to notice is that since we only assign eight HA units to each primary server, the recovery time does not improve as the number of servers increases past nine. At nine, each HA unit can be backed up on its own server. When we fix the scheduling policy, the difference between the random-static placement and dynamic placement is significant, yielding approximately 50% improvement in recovery time. This demonstrates the penalty of a random distribution in that such a distribution will not be balanced in general, and the overall recovery time is bounded by the worst case recovery time across all servers. Moreover, the scheduling algorithm can do better in assigning a checkpoint frequency when the backup servers are well balanced. This has the effect of also reducing recovery time as the checkpoint intervals will get smaller. Notice that min-max improves recovery time by only 25% with random-static, whereas it improves recovery time by 50% with dynamic. Further results on the impact of the processing load as well as the differences in operators can be found in [14].

9 Related Work

Providing high availability through checkpointing has been widely studied in distributed systems. Most approaches rely on stable storage that survives failures [8]. Early approaches conduct coordinated checkpoints so that the stable storage always has a system-wide consistent state [8, 10]. There have been also alternatives that combine asynchronous checkpointing with logging of non-deterministic events [21]. In principle, our approach builds, for each server, a virtual, distributed backup storage on other servers. To maintain this storage, it uses asynchronous, fine-grained checkpoints that are optimized for stream processing.

Modern DBMSs often protect data from server failures by replicating data from a source database, called the primary, to a target database, called the standby. IBM DB2 HADR (High Availability Disaster Recovery) [15], Oracle 10g/DataGuard [18] and MS SQL 2005's Database Mirroring [17] all adopt this style of replication. Workflow systems such as IBM WebSphere MQ [12] also mask server failures by using standby machines. These solutions usually store message logs on a remote site, so that both the primary and standby machines can access them. In contrast to those conventional systems, our solution maintains the backup in a distributed, self-configuring manner and also effectively uses spare CPU cycles to improve system availability.

In the context of our target stream processing domain, a few recent work addressed high availability and related issues (we also discuss them in Sections 2.1 and 2.2). The approaches presented in [19] and [5] rely on "active" HA models in which the backup servers also actively process queries in parallel with the primaries. These active approaches usually have better failover performance, while trading off their query processing capability (note that the backup processes in this case must use the same amount of system resources as the primary processes). We presented a checkpoint-based approach in [13]. Our previous work does not consider cooperative backup and recovery and, therefore, also does not address many of the related challenges and opportunities.

10 Conclusion

This paper considers a novel checkpoint-based HA solution that addresses the needs of distributed stream processing through a parallel fine-grained backup and recovery approach that incurs lower overhead and yields shorter recovery times than existing ones. The key idea is to sub-divide the query at a given server into units that can each be backed-up on a different server. The approach has the advantage that each unit can be checkpointed separately and independently, thereby spreading out the checkpoint burden over time. It also reduces the overall recovery time because each unit can be rebuilt in parallel, making the total recovery time equal to the recovery time of the slowest backup piece.

In this context, we studied how to distribute the backup load in order to minimize the expected recovery time. We also showed how our min-max algorithm adapts to changes in system processing load and performs significantly better than more standard approaches.

References

[1] http://ita.ee.lbl.gov/html/contrib/LBL-TCP-3.html/.

[2] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proc. of the 2st CIDR*. ACM Press, 2005.

[3] 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.

[4] 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.

[5] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proc. of the 2005 ACM SIGMOD*, June 2005.

[6] 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.

[7] 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.

[8] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[9] 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.

[10] 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.

[11] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[12] M. Hiscock and S. Gormley. Understanding high availability with websphere mq. An IBM white paper, May 2005.

[13] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of the 21th ICDE*, Mar. 2005.

[14] J.-H. Hwang, Y. Xing, U. Çetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. Technical Report CS-06-11, Brown University, 2006.

[15] IBM. Ibm db2 - online documentation. http://www.ibm.com/software/data/db2/udb/hadr.html.

[16] C. Mohan and I. Narang. An efficient and flexible method for archiving a data base. In *Proc. of the 1993 ACM SIGMOD*, May 1993.

[17] M. Otey and D. Otey. Choosing a database for high availability: An analysis of sql server and oracle. A Microsoft white paper, Apr. 2005.

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

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

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

[21] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[22] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of the 29th VLDB*, Aug. 2003.

[23] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proc. of the 32th VLDB*, Sept. 2006.

[24] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proc. of the 21th ICDE*, Mar. 2005.