# Shrink – Prescribing Resiliency Solutions for Streaming

Badrish Chandramouli and Jonathan Goldstein
Microsoft Research
{badrishc, jongold}@microsoft.com

## ABSTRACT

Streaming query deployments make up a vital part of cloud oriented applications. They vary widely in their data, logic, and statefulness, and are typically executed in multi-tenant distributed environments with varying uptime SLAs. In order to achieve these SLAs, one of a number of proposed resiliency strategies is employed to protect against failure. This paper has introduced the first, comprehensive, cloud friendly comparison between different resiliency techniques for streaming queries. In this paper, we introduce models which capture the costs associated with different resiliency strategies, and through a series of experiments which implement and validate these models, show that (1) there is no single resiliency strategy which efficiently handles most streaming scenarios; (2) the optimization space is too complex for a person to employ a "rules of thumb" approach; and (3) there exists a clear generalization of periodic checkpointing that is worth considering in many cases. Finally, the models presented in this paper can be adapted to fit a wide variety of resiliency strategies, and likely have important consequences for cloud services beyond those that are obviously streaming.

## 1 INTRODUCTION

Streaming query deployments make up a vital part of cloud oriented applications, like online advertising, online analytics, and internet of things scenarios. They vary widely in their data, logic, and statefulness, and are typically executed in multi-tenant distributed environments with varying uptime SLAs (i.e. how often query response time is impacted by failure). In order to achieve these SLAs, one of a number of proposed resiliency strategies is employed to protect against failure.

Unfortunately, the choice of resiliency strategy is highly challenging, and scenario dependent. For instance, consider the system described in MillWheel [13]. This system periodically checkpoints the query state, and optionally allows users to implement caching, which is highly useful for scenarios like online advertising. In such scenarios, the event rate is small to moderate (e.g., tens of thousands of events per second), and there are a very large number of states (e.g., one for each browsing session) which are active for a short period of time, then typically expire after a long holding period. Rather than redundantly store states in RAM, states are cached in the streaming nodes for a period, then sent to a key-value store after some time, where they are written in replicated fashion to cheap storage, and typically expire unaccessed. As a result, the RAM needed for streaming nodes is small, and may be checkpointed and recovered cheaply.

This design would, however, be untenable for online gaming, where the event rate is high (e.g., millions of events per second), with a large number of active users, and with little locality for a cache to leverage. The tolerance for recovery latency is very low, making it impossible to recover a failed node quickly enough.

While many streaming resiliency strategies are discussed in the literature, along with some modeling work, the state of the art does not quantify the performance and cost tradeoffs across even basic strategies in a way which is actionable in today's cloud environments. For instance, prior efforts (e.g. [11]) do not consider uptime SLAs and resource reservation costs, leading to analyses useful for establishing some intuition for the differences between approaches, but not for selecting strategies in today's datacenters.

Lacking tools or frameworks sufficient to prescribe resiliency approaches, practitioners typically choose the technique which is easiest to implement, or in cases like MillWheel, build systems tailored to solve particular classes of problems, hoping that these systems will have high general applicability.

This paper presents an analytical framework based on uptime SLAs and resource reservation, as well as detailed analyses of a number of resiliency designs for streaming systems. We show:

- ***One size doesn't fit all***: There is no resiliency strategy which efficiently covers most of the streaming query space. Specific strategies can be vastly better compared to others (by orders of magnitude!), depending on scenario and environment characteristics, even when considering only realistic scenarios. While [11] presented similar results for a limited spectrum of strategies, we confirm that this holds across a much broader spectrum of approaches when considering SLAs and with a resource allocation style of provisioning.
- ***No actionable "rules of thumb"***: While some strategies are better than others for specific scenarios, the tradeoffs are too complex for useful "rules of thumb". Models are needed to understand the efficacy of specific approaches for scenarios.
- ***Informative models are tractable***: Models are provided in this paper, and make the alternatives explicit and clear, and, surprisingly, only depend on a few scenario and infrastructure parameters. While our models are a major contribution, it is not necessary that readers understand them in order to use them, or understand the conclusions in this paper.
- ***Our models are accurate***: Using real data and a real streaming system running a real query, we show through our distributed resiliency emulator that the SLAs achieved in practice are typically within 1% of what our models predict.
- ***These models are straightforward to develop***: They can be adapted to describe many resiliency strategies: We provide the precise model modifications for modeling sharded/parallel streaming queries. We also sketch model modifications for handling distributed pipelines and Millwheel style caching.
- ***We introduce active-active periodic checkpointing***: While a generalization of periodic checkpointing, it is not discussed in the literature, likely because it is considered to be inferior to active-active on-demand checkpointing. We show that periodic checkpointing is a better strategy in most situations.

Paper organization: Section 2 describes the modeled resiliency strategies. Section 3 describes our modeling framework, including our metrics and parameters, and our modeling assumptions. Section

4 provides models for 5 resiliency strategies. Section 5 validates the accuracy of our models using a distributed resiliency emulator and a real query on real data with a real streaming query processor. Section 6 evaluates the strategies, by applying our models with varying parameter settings. Section 7 describes extensions to our models which take into account other kinds of resources, and how those models could be applied in situations with sharding or caching. Section 8 gives an overview of related work. Section 9 concludes the paper with lessons learned and future work.

# 2 RESILIENCY STRATEGIES

This section gives an overview of the fundamental resiliency strategies considered in this paper. Note that these are foundational approaches, mostly described in the literature, and can be varied to create derivative solutions like the one used in MillWheel. In Section 7, we discuss these derivative strategies, and how the models in this paper can be applied.

These foundational strategies are described visually in Figure 1, Figure 2, and Figure 3, which show the states that a typical streaming node goes through for different resiliency approaches. These figures will be referred to throughout this section. Note that initially, we do not consider sharded scenarios. We relax this restriction with precise model modifications in Section 7.

In the figures below, nodes begin by recovering the state of the failed node which they are replacing. This is the case for all nodes except for nodes which initially start the query. Similarly, the lifetimes of almost all streaming nodes end with failure.

Note that in all resiliency approaches described in this paper, we assume the existence of a resilient store, and further assume that all input is journaled in this store. Furthermore, for all cases, except one version of replay based (for explanatory purposes), we assume that all output must be delivered exactly once in the face of failure.

## 2.1 Replay Based

These scenarios leverage knowledge of the query's window size. For instance, in a 1 minute trailing average, the window size is 1 minute. Note that such information isn't always available, in which case these resiliency approaches are not possible.

In the single node version, as described by the timeline in Figure 1, when the node fails, a new node is created which first consumes a window of input. During this time, the query falls further behind, so it subsequently enters a catchup phase until normal operation can resume. Note that one can either start consuming input from a point in time which guarantees no loss of output, or choose a moment in time a bit later which minimizes catch up time.

In active-active replay, all nodes simultaneously run the query. When a copy fails, it recovers in the same manner as single node replay. The query is only down when all running copies go down. Active-active approaches are critical for meeting tough SLAs, but how many copies should be run for a given scenario and SLA?

Note that for all active-active approaches, including replay based, we assume that there is a primary copy which is responsible for sending output. Part of handling failure is to seamlessly switch primaries from one copy to another. As a result, the cost of output transmission doesn't vary significantly between strategies.

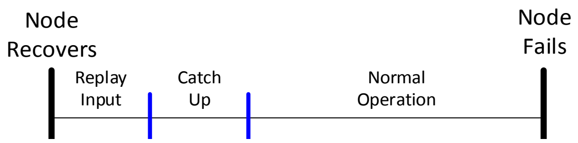

**Figure 1: Replay Based Node Timeline**

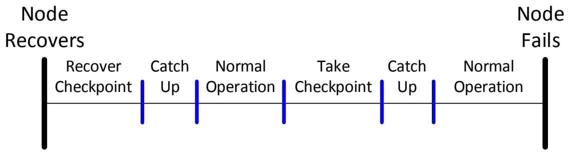

**Figure 2: Periodic Checkpointing Based Timeline**

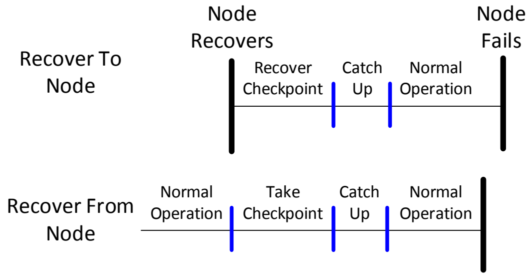

**Figure 3: On Demand Checkpointing Based Timeline**

## 2.2 Periodic Checkpointing Based

These solutions make use of some systems' ability to checkpoint the state of a running query. As shown in Figure 2, the running query periodically checkpoints its state to a resilient store. Upon failure, the latest checkpoint is read and rehydrated on a new node, and the input is replayed from the time of the checkpoint. Note that for checkpointing based strategies, duplicate output is typically thrown away as part of catching up [5].

While we found no reference to the active-active version, it is a clear extension of the single node version, where the query is run on many nodes. One reserved copy periodically checkpoints. When a copy fails, a new copy is spun up as in the single node version. If the checkpointing node fails, during the subsequent catchup phase, all checkpoints are taken at the correct logical times according to the checkpointing period, as if the node wasn't recovering.

Checkpointing based solutions are typically chosen when either replay solutions aren't possible, or where the checkpoint size is significantly smaller than the input needed to reproduce it, but how much smaller does the checkpoint need to be? Are there other important factors?

## 2.3 On-demand Checkpointing Based

These are the solutions usually referred to in the literature as active-active checkpointing. As shown in Figure 3, in this approach, multiple copies of the computation are run. When a node fails, another running node stops processing input and takes a checkpoint, which is used to rehydrate a new running copy. Note that this approach requires at least 2 running nodes.

This approach never writes checkpoints to storage, checkpoints only when needed, and catchup times are less. However, an extra node is needed to jump-start a failed node (i.e., when a node goes down, two stop processing input), and if all running copies fail, the state is lost. As we will see, in practice, this strategy is mostly inferior to active-active periodic checkpointing.

# 3 MODELING FOUNDATIONS

In this section we formalize the metrics by which we evaluate resiliency approaches, and the parameters upon which they depend. We begin with a general discussion of the challenges of deciding on the metrics of interest, and conclude the section with a precise statement of the chosen metrics and parameters.

## 3.1 Modeling Objectives

Typically, streaming queries are run on one or more nodes in a datacenter, and incur various costs, including:

- CPU costs to run, recover, and checkpoint the query
- Storage costs to resiliently journal the input and checkpoints
- Networking costs to move input and checkpoints.
- Memory costs associated with maintaining query state

All of these costs are affected by the choice of resiliency strategy, whose goal is to meet a **downtime SLA**. This type of SLA allows the user to specify, for instance, the number of minutes per year during which the query is "down". Down, in this context, means that the results are not being delivered in as timely of a fashion as they would if failure didn't occur. For instance, if a query is catching up after failure and recovery, this is considered downtime until the query has completely caught up to the arriving input.

In this paper, we specifically model **NIC bandwidth** costs as a proxy for overall network costs. This choice captures all network activity at the edges, regardless of internal topology, including network capacity to and from storage nodes, compute nodes, and ingress nodes. Our models capture the complexity present in modeling other resources, and can be varied to capture the other resource costs (see Section 7.1).

By only considering NIC costs in the models presented and evaluated here, we will miss some phenomena. For instance, when states are hard to compute (i.e. high computational complexity), checkpointing is typically better than replay. Also, sometimes memory is a critical cost which can affect the choice of resiliency strategy, as was the case in MillWheel. Section 7 sketches how our models can be extended to analyze these cases. Note that making our models sensitive to these phenomena results in a more complex, but still tractable, optimization space, which strengthens our conclusion that an actual model is needed.

In order to compute cost, we take a **bandwidth reservation** approach. More specifically, consider the network load profiles of compute nodes for the three types of resiliency approaches, shown in Figure 4. For all scenarios, each query on a node begins its life by recovering a previously failed query's state.

For replay, once recovery is complete, the load settles down to the same load that would exist without resiliency. This suggests that we must find enough bandwidth on the node to recover quickly enough to meet our SLA, but that we can significantly lower the bandwidth requirements once recovery is over, leaving room on the node for other work.

For periodic checkpointing based approaches, there is one node which periodically checkpoints. For the single node version, if enough bandwidth isn't available for either recovery or checkpointing, we will not be able to meet our SLA. Since increasing the bandwidth reservation of a node could be heavily disruptive to other jobs on the node, resulting in SLA failure for those jobs, we reserve enough capacity to accommodate recovery initially, and periodic checkpointing until failure, even though there are periods, after recovery and between checkpoints, where the network load is lower. This requirement that reservations only decrease over time is met for all reservation strategies.

For active-active periodic checkpointing, since the checkpointing node is never used for output, it may fall behind without impacting the SLA, since other nodes, which aren't checkpointing, are always up to date. Rather, the checkpointing node must keep up overall with a constant reservation for the average needed bandwidth, but may fall behind for periods of time. We therefore need to only reserve the average, rather than the peak, load. Note that some nodes will never need to checkpoint. These nodes have load profiles like the replay based approach, and we can similarly decrease their bandwidth reservation once recovery is over.

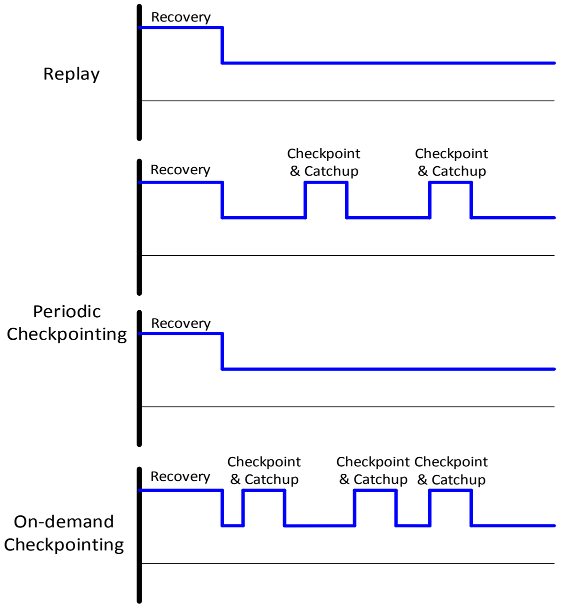

**Figure 4: NIC Load for Compute Nodes**

For on-demand based approaches, after recovery is over, any node may, at any time, be used to start a new instance. The load is therefore characterized by sporadic heavy load associated with checkpointing. Like single node periodic checkpointing, we continuously reserve the peak checkpointing load needed to ensure that the SLA is met.

With this bandwidth reservation approach in mind, the goal of our model is to answer two questions:

1. How much bandwidth, compared to the input bandwidth, do I need to reserve initially to recover my query?
2. How costly, in terms of reserved NIC bandwidth, is my resiliency approach compared to running the query non-resiliently?

Note that both of the costs mentioned above are **in comparison to the cost of running the query non-resiliently**. This is a deeply important, nonobvious facet of our modeling approach which greatly simplifies our modeling task.

## 3.2 Modeling Assumptions

In order to simplify our analysis, we make certain assumptions:

- All network load and other work associated with processing the query unresiliently is unvarying over time. As a result, these models are useful for capturing "worst cases". This assumption is deeply embedded in our approach and cannot be relaxed without deep changes in our models.

- The output is small compared to the input and is, therefore, not part of the model. This assumption simplifies our presentation, and is almost always true for streaming queries. Output transmission could easily be added to our models.
- Failure doesn't occur during recovery. This is an assumption made to simplify the presentation of our models. In all cases, this is a second order effect, and only has small impact on the resulting costs. This assumption could be relaxed by extending the presented approaches.

## 3.3 Modeling Metrics and Parameters

With the above two goals in mind, we introduce the following two metrics which will be computed for each resiliency option, given application and infrastructure parameters:

- $R_F$ = The recovery NIC bandwidth reservation needed to meet the SLA, as a factor of input bandwidth. The subscript refers to the unit (factor).
- $C_F$ = The cost, in terms of total reserved NIC bandwidth, as a factor of the NIC costs associated with running the query non-resiliently (factor).

These metrics are computed using these application parameters:

- $W_T$ = windows size, such as 10 minutes in a 10 minute trailing window (time, e.g. sec).
- $C_S$= Checkpoint size (size, e.g. bytes)
- $I_R$= Input rate (size/time)
- $SLA$ = Fraction of the time that the system response to input is unaffected by failure (ratio, e.g. .99999)

We also have the following infrastructure parameters:

- $F_T$ = Mean time between failure for a single node (time)
- $K_F$ = Number of copies in replicated storage (factor)

In addition, there are, for some resiliency strategies, the following tunable parameters, which we set as part of optimizing for cost:

- $C_T$ = Checkpoint period for periodic checkpointing (time)
- $N_F$ = Number of running copies (factor). This is either explicitly set, or varied as part of optimizing cost.

Finally, there is, throughout the following analyses, the following computed value, which is computed from the above parameters:

- $S_T = \frac{C_S}{I_R}$–= The checkpoint transfer time assuming input rate bandwidth (time)

## 3.4 Computing $C_F$

Figure 5 illustrates the network flows when computation isn't resilient to failure. The NIC costs associated with these flows form the baseline with which all other approaches are compared.

First, note that the data could initially be acquired by the ingress node with a network flow arriving at the node, although the data could also be born at this node. Also, there is a network flow transmitting the input to the compute node, as well as a network flow to each of the storage nodes on which a copy of the data will be stored. Note that there is only one path on the ingress node to all the storage nodes which store the data. This reflects our decision to capture the costs in common with all implementations of cloud storage. All implementations must push a copy to each of $k$ storage nodes, but whether internal network communication is reduced with interesting topologies and/or broadcast networks varies amongst implementations. These varying costs could easily be accounted for in all of our models for a particular storage implementation. Also, note that the storage nodes in our figure are logical, as a single copy of the data may actually be spread out over a very large number of nodes in a storage cluster. The aggregate NIC bandwidth is, however, insensitive to this, so we represent each of these copies as sent to a single node.

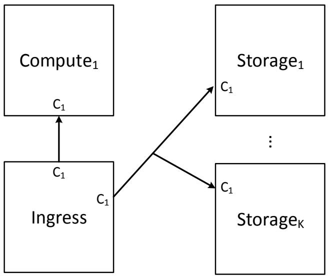

**Figure 5: Network Flow Diagram With No Resiliency**

Associated with each network flow are NIC costs at either end (i.e., $C_1$), which, in this case, are symmetric. For some strategies, however, the reservations are not symmetric, and for this reason, we separately account for the costs at both ends. To compute the cost of all network flows, we calculate the **expected total NIC reservation costs during a failure period, as a factor of the input rate**. As a result, $C_1 = F_T$ is used here, since there is an input rate sized NIC bandwidth reservation for $F_T$ time units.

When computing $C_F$, we will not include the cost of acquiring the data, since it is insensitive to the choice of resiliency strategy, and the data may be born on the node, in which case there are no network costs. As a result, the baseline cost, adding up all the network flow costs at both sender and receiver, is $2 \cdot F_T$ for the ingress node, $F_T$ for the compute node, and $k \cdot F_T$ for the storage nodes, or $(k + 3) \cdot F_T$ in total. This will be used as the denominator in every cost calculation throughout the paper.

# 4 RESILIENCY MODELS

We now describe five of our models. Specifically, we model the single node versions of replay and periodic checkpointing, and the active-active variants of replay, periodic checkpointing, and on-demand checkpointing.

## 4.1 Single Node Replay

There are two versions of single node replay. In the first, we assume that lost output is acceptable, and that the goal of recovery is to minimize downtime. This assumption is desirable for dashboards, where users are frequently uninterested in previous results (e.g. task manager in windows). We also provide an analysis where all output is computed. This is desirable when output is logged, or where visualizations provide the history of a reported metric.

We begin our first analysis by deriving $R_F$. Note that we are trying to find the minimal setting for $R_F$ which meets our SLA over an arbitrarily long period of time. In particular, to exactly meet our SLA in the long run, each failure is allowed a downtime budget, which, on average, is used to fully recover when the query initially starts. In particular, that budget:

$$B_T = F_T \cdot (1 - SLA)$$

Observe that for each failure, the recovery time $R_T$ is:

$$R_T = \frac{W_T}{R_F}$$

This can be seen when one considers that to recover, one must replay exactly a window's worth of data. If the bandwidth reservation is double what is needed to process the input in real time, recovery can happen in half the window size period of time, and so on. We can now set $R_T = B_T$, and we have:

$$F_T \cdot (1 - SLA) = \frac{W_T}{R_F},$$

$$R_F = \frac{W_T}{F_T \cdot (1 - SLA)}$$

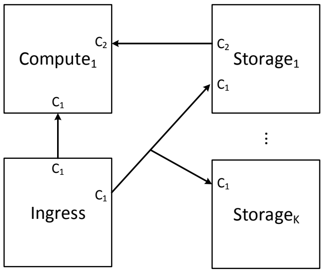

**Figure 6: Network Flow Diagram for Single Node Replay**

In computing $C_F$, first consider Figure 6, which shows the flows and costs associated with replay. First, note that cost $C_1$ is the same as in the non-resilient case. We additionally have cost $C_2$, which is associated with the replay flow. This cost is $R_T \cdot R_F$, minus the cost of the portion of replay which involved receiving data for the first time from the ingress node, or $R_T$. Resulting in:

$$C_2 = R_T \cdot R_F - R_T$$

Summing up all the costs:

$$C_F = \frac{2 \cdot (R_T \cdot R_F - R_T) + (K_F + 3) \cdot F_T}{(K_F + 3) \cdot F_T}$$

On the other hand, if we are not allowed to miss output, then recovery time must start reading input starting from a full window before failure occurred. Once a full window of data has been read, we have fallen behind by the time it took to transmit that window's worth of data. Once we have caught up by that amount, we have further fallen behind by a smaller amount, and so on. This leads to the following infinite series:

$$R_T = \frac{W_T}{R_F} + \frac{W_T}{R_F^2} + \frac{W_T}{R_F^3} + \cdots$$

Note that, for convenience, we will frequently substitute:

$$U = \frac{1}{R_F}$$

We now have the geometric series:

$$R_T = W_T \cdot U \cdot \sum_{i=0}^{\infty} U^i, U < 0$$

Using the closed form for the series, we get:

$$R_T = \frac{W_T \cdot U}{(1 - U)} = B_T = F_T \cdot (1 - SLA)$$

$$U = \frac{F_T \cdot (1 - SLA)}{W_T + F_T \cdot (1 - SLA)}$$

$$R_F = \frac{W_T + F_T \cdot (1 - SLA)}{F_T \cdot (1 - SLA)}$$

The total cost is more straightforward to calculate, as $C_2$ is just the cost of reading a window's worth of data. Thus:

$$C_F = \frac{2 \cdot W_T + (K_F + 3) \cdot F_T}{(K_F + 3) \cdot F_T}$$

## 4.2 Single Node Periodic Checkpointing

In this resiliency approach, checkpoints are taken periodically. Both the downtime experienced during checkpointing, as well as the downtime experienced during recovery are charged against the downtime budget.

We start with the downtime experienced during checkpointing: Every failure period, $\frac{F_T}{C_T}$ checkpoints are taken, each of which takes $S_T$ time units to transfer over the network, assuming input rate bandwidth. In addition, there is a catch up period after each checkpoint is taken, which is the time it takes for the output to be produced in as timely a fashion as if a checkpoint had never been taken. Clearly, the amount of time it takes for the checkpoint to be transferred is $U \cdot S_T$.

The catch up time is a bit trickier: during the time it took to take the checkpoint, the input fell behind by $U \cdot S_T$ time units. It takes $U \cdot U \cdot S_T$ time to replay this input, at the end of which, we are now behind by $U \cdot U \cdot U \cdot S_T$ time units. In other words, the catch up time can be expressed with the following geometric series: $U \cdot U \cdot S_T \cdot \sum_{i=0}^{\infty} U^i$. Since the only way we can catch up is if $U < 1$, the closed form for the series can be used, and the downtime cost of checkpointing for each failure period, $B1_T$, can be written as:

$$B1_T = U \cdot \left(S_T + \frac{U \cdot S_T}{1 - U}\right) \cdot \frac{F_T}{C_T}$$

Calculating the downtime associated with recovery is a bit more difficult. Similar to taking a checkpoint, there are two phases: a checkpoint recovery period, and a catch up period. Unlike taking a checkpoint, the catch up period depends upon how long ago a checkpoint was taken.

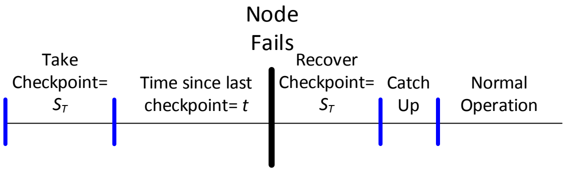

**Figure 7: Periodic Checkpointing Recovery Timeline**

Consider Figure 7, which shows the timeline for checkpointing and recovery. Observe that on the far left, the last completed checkpoint begins. $t$ time units after the checkpoint is successfully transferred, failure occurs. At this time, checkpoint recovery begins, and the checkpoint is transferred. Once the transfer is complete, catch up commences.

The time to recover the checkpoint after recovery begins is clearly $U \cdot S$. The recovery time has two components. The first is a "fixed" component that doesn't vary with the amount of time since the last checkpoint was successfully transferred. Another way to look at this is that this is the catchup time if $t = 0$. In this case, the total amount of input which needs to be replayed is the time it took to transfer the checkpoint when it was taken, plus the time it took to recover the checkpoint after failure. Thus the total amount of fixed input time which needs to be recovered is $2 \cdot U \cdot S_T$. Note that we still have the infinite sum as we replay, so the total budget used for the fixed replay cost is $\frac{2 \cdot U \cdot U \cdot S_T}{1-U}$. In addition to this, there is a variable replay amount, $t$, which varies from 0 to $C_T$. We account for this variable cost by doing an expected value calculation for $t$. Therefore the total replay cost $B2Replay_T$, is:

$$B2Replay_T = U \cdot \left( \frac{2 \cdot U \cdot S_T}{1 - U} + \frac{\int_0^{C_T} t \cdot dt}{(1 - U) \cdot C_T} \right)$$

It follows directly that the total recovery cost, $B2_T$, which includes both the cost of restoring the checkpoint, and the replay cost, is:

$$B2_T = U \cdot \left( S_T + \frac{2 \cdot U \cdot S_T}{1 - U} + \frac{\int_0^{C_T} t \cdot dt}{(1 - U) \cdot C_T} \right)$$

The total cost per failure, $B_T$, is therefore:

$$B_T = B1_T + B2_T = U \cdot \left( S_T + \frac{U \cdot S_T}{1 - U} \right) \cdot \frac{F_T}{C_T} + U \cdot \left( S_T + \frac{2 \cdot U \cdot S_T}{1 - U} + \frac{\int_0^{C_T} t \cdot dt}{(1 - U) \cdot C_T} \right)$$

Note that really we are interested in maximizing $U$, which is equivalent to solving for $U$ when the recovery budget per failure equals the maximum allowable downtime per failure, or:

$$(1 - SLA) \cdot F_T = B_T$$

Throughout our modeling efforts, we are presented with such equations, and while sometimes it is possible to solve for $U$ analytically, in general, we take a numerical approach. For instance, in this case, we find the zero for:

$$F(U) = B_T - (1 - SLA) \cdot F_T$$

Since $F(U)$ is monotonically increasing, $0 < U < 1$ , $F(0) < 0$, and $f(1)$ is an asymptote at infinity, we simply do a binary search between 0 and 1, avoiding any potential instability issues in a technique like Newton's method. Note, though, that we need to find the zero in the above equation for a setting of $C_T$ which optimizes cost. The details of the numerical approach taken to solve this problem are described in Section 4.6. It is worth pointing out, here that this numerical problem is solved very efficiently, and with no numerical stability issues.

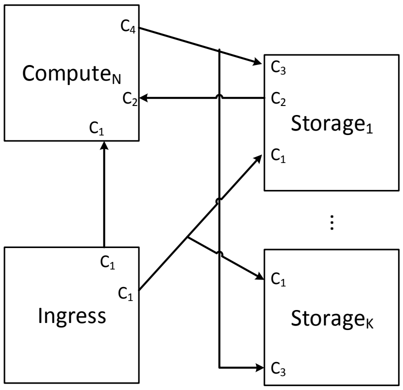

**Figure 8: Network Flow Diagram for Single Node Periodic Checkpointing**

Once we determine $U$, we can compute $R_F$:

$$R_F = \frac{1}{U}$$

In computing $C_F$, first consider Figure 8, which shows the flows and costs associated with single node periodic checkpointing. First, note that cost $C_1$ is the same as in the non-resilient case, incurring a cost of:

$$(K_F + 3) \cdot F_T$$

Determining $C_2$, the network costs associated with recovery, is similarly straightforward, as it includes the cost of sending or receiving a checkpoint, and sending or receiving, on average, half the checkpointing period:

$$C_2 = S_T + \frac{C_T}{2}$$

There are two NICs where $C_2$ cost is incurred, which we will elaborate on further.

Note that on the checkpointing edge, $C_4/C_3$, the network reservation on the compute node, and on each storage node, are different. This is due to our assumption that bandwidth reservations on compute nodes are only allowed to decrease over time, as increasing a reservation may not always be possible without significant disruption of other activity. As a result, not all reserved capacity is actually consumed. On the other hand, storage systems are extremely effective at spreading out load, making is possible to easily adjust to increasing and decreasing bandwidth requirements. As a result, $C_1 + C_4 + C_2 = R_F \cdot F_T$.

$C_3$, on the other hand, is the actual cost of checkpointing, which, similar to active-active checkpointing, is:

$$C_3 = \frac{S_T \cdot F_T}{C_T}$$

We are now ready to write the total cost:

$$C_F = \frac{(K_F + 2) \cdot F_T + (c_1 + c_4 + c_2) + c_2 + K_F \cdot C_3}{(K_F + 3) \cdot F_T} = \frac{(K_F + 2 + R_F) \cdot F_T + S_T + \frac{C_T}{2} + \frac{K_F \cdot S_T \cdot F_T}{C_T}}{(K_F + 3) \cdot F_T}$$

## 4.3 Active-Active Periodic Checkpointing

Recall that with this resiliency approach, multiple copies of the streaming computation are running, and one of these copies is reserved for periodic checkpointing. When one of the copies goes down, recovery from the last successful checkpoint is initiated. As long as at least one non-checkpointing copy remains, there is no downtime. If, after a time, all copies go down, the remaining recovery time is charged against the SLA budget for that failure.

We begin our analysis by describing our failure model for nodes used in active-active approaches. Specifically, assume that the distribution for the amount of time it takes for a node to fail is captured by the exponential distribution [22]. We determine the resiliency cost associated with all running copies failing before recovery is complete, as follows:

Let the random variables $X_i =$ the time for node $i$ to fail given $\lambda = \frac{1}{F_T}$. The PDF and CDF for $x_i, f(t)$ and $F(t)$ respectively, are:

$$f(t) = P(X_i = t) = \lambda e^{-\lambda t}$$

$$F(t) = P(X_i \leq t) = 1 - e^{-\lambda t}$$

Let $Y =$ the time for the $k = N_F - 2$ remaining nodes to fail. The PDF and CDF for $Y$, $g(t)$ and $G(t)$ respectively, are:

$$G(t) = P(Y \leq t) = \prod_{i=1}^{k} P(X_i \leq t) = (1 - e^{-\lambda t})^k$$

$$g(t) = \frac{d(G(t))}{dt} = \frac{d((1 - e^{-\lambda t})^k)}{dt}$$

Each time a node fails, its state must by recovered and the node must be caught up to the latest input. If all other nodes fail before recovery is complete, then the user will experience downtime, which will be charged against the downtime budget.

We now consider the impact to our resiliency budget in 3 cases. In all these cases, t is the time until all running nodes fail after one begins recovery. Recovery involves both a fixed sized cost, which includes the time to recover the checkpoint, and an input catch up cost which is twice the time it takes to take a checkpoint (time to take the checkpoint and time to restore the checkpoint), plus an additional variable sized input catch up cost, which depends on how far back the last checkpoint completed.

### 4.3.1 Case 1: $t < U \cdot \left(S_T + \frac{2 \cdot U \cdot S_T}{1-U}\right)$

In this case, failure occurs before the fixed portion of the recovery cost is complete. This includes the time to restore a checkpoint of time length $S_T$, plus the time length of input which arrived while the used checkpoint was taken (i.e. $U \cdot S_T$), plus an equal amount of input which arrived while the checkpoint was restored.

Consider a variable $0 < p < C_T$, which represents, at the time of initial failure, the amount of time which passed since the last checkpoint completed. For a given t, the budget used is:

$$b1_T(t) = \int_0^{C_T} \frac{U \cdot \left(S_T + (2 \cdot U \cdot S_T + p) \cdot \sum_{i=0}^{\infty} U^i\right) - t}{C_T} dp$$

$$= \int_0^{C_T} \frac{U \cdot \left(S_T + \frac{2 \cdot U \cdot S_T}{1-U} + \frac{p}{1-U}\right) - t}{C_T} dp$$

Note that in the above, $U \cdot S_T$ is the portion of recovery associated with rehydrating the checkpoint, while $U \cdot (2 \cdot U \cdot S_T + p) \cdot \sum_{i=0}^{\infty} U^i$ is the time needed to catch up, depending on how long it's been since the last checkpoint completed. The $2 \cdot U^2 \cdot S_T$ portion of this reflects the time to catch up associated with both taking and restoring the checkpoint.

Integrating over the relevant times for this case, the overall impact on our recovery budget is:

$$B1_T = \int_0^{U \cdot \left(S_T + \frac{2 \cdot U \cdot S_T}{1-U}\right)} g(t) \cdot b1_T(t) \cdot dt$$

### 4.3.2 Case 2:

$$\mathrm{U} \cdot \left(\mathrm{S_T} + \frac{2 \cdot \mathrm{U} \cdot \mathrm{S_T}}{1 - \mathrm{U}}\right) < \mathrm{t} < \mathrm{U} \cdot (\mathrm{S_T} + \frac{2 \cdot \mathrm{U} \cdot \mathrm{S_T}}{1 - \mathrm{U}} + \mathrm{C_T} \cdot \sum_{\mathrm{i}=0}^{\infty} \mathrm{U^i})$$

Or equivalently:

$$\mathrm{U} \cdot \left(\mathrm{S_T} + \frac{2 \cdot \mathrm{U} \cdot \mathrm{S_T}}{1 - \mathrm{U}}\right) < \mathrm{t} < \mathrm{U} \cdot \mathrm{S_T} + \frac{\mathrm{U} \cdot (2 \cdot \mathrm{U} \cdot \mathrm{S_T} + \mathrm{C_T})}{1 - \mathrm{U}}$$

In this case, failure happens after all fixed recovery costs, but we cannot conclude that recovery completes in all cases before total failure occurs. For each value of $t$ in this range, there are some sub-cases where total failure occurs before catch-up is complete, which incurs a cost against our resiliency budget, but there are also some sub-cases where total failure occurs after catch-up is complete, incurring no penalty.

In particular, in the above upper bound, $U \cdot S_T$ represents the time to rehydrate the checkpoint, while the second term, $\frac{U \cdot (2 \cdot U \cdot S_T + C_T)}{1-U}$, represents the portion of the recovery time to catch-up, by as much as $U \cdot (2 \cdot U \cdot S_T + C_T)$ after checkpoint rehydration is complete.

Consider a variable $t_p = t - U \cdot \left(S_T + \frac{2 \cdot U \cdot S_T}{1-U}\right)$, which represents how much time we had after the fixed portion of the recovery, to catch up before total failure. Furthermore, consider a scaled version of $p$, called $p_c$, which is the amount of variable catch-up time needed given a particular value of $p$. Note that:

$$p_c = \frac{U \cdot p}{1 - U}$$

Consider Figure 9, which illustrates the entire range of possibilities for the current case. For each time $t_p$, we have enumerated the space of possibilities, which is to say, that $p$ could range anywhere from 0 to $C_T$, resulting in:

$$0 \leq p_c \leq \frac{U \cdot C_T}{1 - U}$$

Now consider the diagonal where $t_p = p_c$. This is the case where the new node exactly catches up when the last running node fails, resulting in 0 downtime. For the lower right triangle, the new node has been fully caught up before the other nodes fail, also resulting in 0 downtime. There are also contour lines, parallel to and above the diagonal, which represent constant and increasing amounts of time between catch-up and failure. We now define a new variable $x = p_c - t_p$. In order to calculate the contribution of these scenarios to the cost of resiliency, we calculate:

$$B2_T = \int_0^{\frac{U \cdot C_T}{1-U}} x \cdot P(X = x)\, dx$$

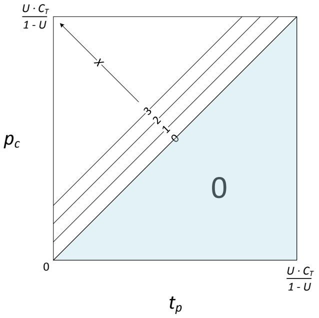

**Figure 9: Case 2 variables and integration limits**

In other words, we sum the various cost contour lines, where each contour is multiplied by the likelihood of occurrence for that contour. In order to calculate the likelihood, we integrate across the relevant range of $t_p$, summing the probabilities of all points along the contour line. We are aided here by the assumption that when failure occurs, there is a uniform probability distribution (between 0 and $C_T$) for how far back the last checkpoint completed. Thus:

$$P(X = x) = \int_{t_p=0}^{\frac{U \cdot C_T}{1-U} - x} g\left(t_p + U \cdot \left(S_T + \frac{2 \cdot U \cdot S_T}{1 - U}\right)\right) \cdot \left(\frac{1}{\left(\frac{U \cdot C_T}{1 - U}\right)}\right) dt_p$$

$$= \int_{t_p=0}^{\frac{U\cdot C_T}{1-U}-x} \left( \frac{g\left(t_p + U\cdot\left(S_T + \frac{2\cdot U\cdot S_T}{1-U}\right)\right)\cdot(1-U)}{U\cdot C_T} \right) dt_p$$

A few notes:

- The $t_p$ in $g\left(t_p + U\cdot\left(S_T + \frac{2\cdot U\cdot S_T}{1-U}\right)\right)$ comes from the outer integral. We add the fixed costs of recovery to t because we are picking probabilities for total failure times which occur after these costs are incurred (i.e. we are converting from $t_p$ to $t$). We divide the resulting probability to spread it out uniformly amongst all the cases for that total failure time.
- The upper bound on the definite integral decreases as $x$ increases because we only integrate the portion of the contour line below $p_c = \frac{U\cdot C_T}{1-U}$ . As we increase $x$, the portion of the contour line we integrate over therefore gets shorter.

Thus, the total contribution of this case to our resiliency budget is:

$$B2_T = \int_0^{\frac{U\cdot C_T}{1-U}} x\cdot \left( \int_{t_p=0}^{\frac{U\cdot C_T}{1-U}-x} \left( \frac{g\left(t_p + U\cdot\left(S_T + \frac{2\cdot U\cdot S_T}{1-U}\right)\right)\cdot(1-U)}{U\cdot C_T} \right) dt_p \right)\cdot dx$$

### 4.3.3 Case 3: $t > U\cdot S_T + \frac{U\cdot(2\cdot U\cdot S_T + C_T)}{1-U}$

In this case failure is guaranteed to occur after recovery is complete, and there is no impact on our resiliency budget. Therefore:

$$B3_T = 0$$

Considering all cases, the overall resiliency cost per failure is:

$$B_T = B1_T + B2_T + B3_T$$

Our goal is to solve for $U$ in:

$$(1 - SLA)\cdot\frac{F_T}{N_F} = B_T$$

First, note the use of $N_F$ in calculating our per failure budget. Our budget is adjusted thus because failure is more common by a factor of $N_F$, reducing the per failure budget. As in the single node periodic checkpointing case, we take a numerical approach. Specifically, we find the zero for:

$$F(U) = B_T - (1 - SLA)\cdot\frac{F_T}{N_F}$$

Again, since $F(U)$ is monotonically increasing, $0 < U < 1$, $F(0) < 0$, and $f(1)$ is an asymptote at infinity, we simply do a binary search between 0 and 1, avoiding any potential instability issues in a technique like Newton's method. Note that in practice, we must solve the above equation with values chosen for $C_T$ and $N_F$ which optimize cost. Since $N_F$ has a small number of finite values worth considering, we optimize cost for each distinct value of $N_F$, each of these optimization problems are solved in a manner similar to single node checkpointing, with the specific numerical approach described in Section 4.6.

Once we determine $U$, we can compute $R_F$:

$$R_F = \frac{1}{U}$$

In computing $C_F$, first consider Figure 10, which shows the flows and costs associated with active-active periodic checkpointing. First, note that cost $C_1$ is the same as in the non-resilient case, although there are additional flows with these costs due to the active-active nature of this solution, incurring costs of:

$$(K_F + 1 + 2\cdot N_F)\cdot F_T$$

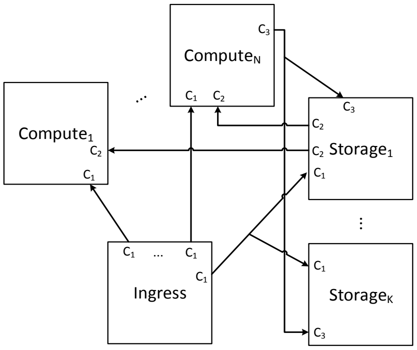

**Figure 10: Network Flow Diagram for Active-Active Periodic Checkpointing**

We additionally have cost $C_2$, which is associated with the recovery flow, and occurs, on average, $N_F$ times during $F_T$. This flow consists of sending and receiving a checkpoint, followed by catching up to the point of failure by replaying stored input. Since the expected time since the last checkpoint is $C_T/2$, the total costs associated with $C_T$ are:

$$N_F\cdot(2\cdot(S_T + \frac{C_T}{2}))$$

$C_3$, the network costs of taking a checkpoint, like $C_2$, involves sending and receiving checkpoints, except that there is no replay component, it occurs $F_T/C_T$ times during the failure interval, and is sent to $K_F$ storage nodes, leading to a cost of:

$$\frac{(K_F + 1)\cdot S_T\cdot F_T}{C_T}$$

Summing all the components of $C_F$ leads us to the following:

$$C_F = \frac{(K_F + 1 + 2\cdot N_F)\cdot F_T + N_F\cdot 2\cdot(S_T + \frac{C_T}{2}) + \frac{(K_F + 1)\cdot S_T\cdot F_T}{C_T}}{(K_F + 3)\cdot F_T}$$

## 4.4 Active-Active On-Demand Checkpointing

### 4.4.1 Analysis I

In this approach, during normal operation, the query is redundantly executed $N_F$ times. When a copy goes down, one of the remaining copies is used to spin up a new copy while all other copies continue. The new copy is created as in the previous on demand checkpointing scenario. This results in reduced time to recover, compared to periodic checkpointing, and eliminates the overhead associated with periodic checkpointing. Note, however, that there is one less redundant node available during recovery since one node is reserved for spinning up another copy. This makes it more likely that all nodes will go down before the first node recovers, incurring more frequent charges to the SLA budget.

When a node fails, another node takes a checkpoint, and sends the checkpoint to a new node, which rehydrates the checkpoint. Since all operations can be pipelined, the time taken to simultaneously transfer and receive the checkpoint is $U\cdot S_T$. During this time, both nodes will fall behind and will need to catch up. As a result, the total amount of recovery time is:

$$R_T = U\cdot\left(S_T + \frac{U\cdot S_T}{1-U}\right)$$

Similar to active-active with replay, given the recovery time, the impact on the resiliency budget is:

$$B_T = \int_0^{U\cdot\left(S_T+\frac{U\cdot S_T}{1-U}\right)} g(t)\cdot\left(U\cdot\left(S_T+\frac{U\cdot S_T}{1-U}\right)-t\right)\cdot dt$$

One disadvantage of this approach is that the function $g(t)$ is calculated based upon $N_F - 2$ nodes that must all fail instead of $N_F - 1$, as used in other active-active approaches.

$U$ may now be numerically calculated, as in other approaches, and:

$$R_F = \frac{1}{U}$$

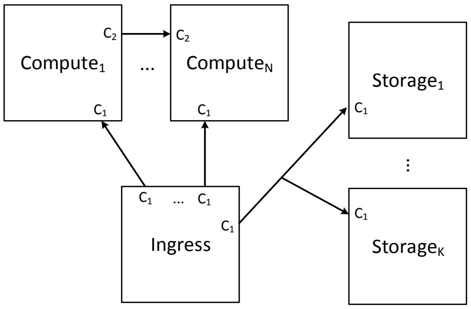

**Figure 11: Network Flow Diagram for Active-Active On-Demand Checkpointing**

In computing $C_F$, first consider Figure 11, which shows the flows and costs associated with active-active on-demand checkpointing. Note that all networking costs for compute nodes are captured by making use of $R_F$. To this we add the remaining $C_1$ costs, yielding:

$$C_F = \frac{(K_F + 1 + N_F)\cdot F_T + N_F \cdot R_F \cdot F_T}{(K_F + 3)\cdot F_T}$$

### *4.4.2 Analysis II*

Note that the first analysis assumes that neither the recovering node nor the recovered node fail during recovery. This avoids the catastrophic case that all nodes fail during recovery, which results in a complete loss of state. Therefore, in this analysis, we determine the level of provisioning required to ensure that the expected time for complete failure is equal to some controlled amount (e.g. 10 years).

Similar to previous cases, we use the same definitions for $G(t)$ and $g(t)$, since we are interested in the likelihood that all nodes fail within the recovery period. In other words, we are interest in $G(R_T)$. Note that we know $R_T$ from Analysis I:

$$G(R_T) = \left(1 - e^{-\frac{U\cdot\left(S_T+\frac{U\cdot S_T}{1-U}\right)}{F_T}}\right)^k$$

Therefore, for each failure, there is a $G(R_T)$ chance that all nodes will fail before recovery of this failure is complete. Considering each failure as a Bernoulli trial, we can use the negative binomial distribution, which is the discrete probability distribution for the number of successes in a sequence of independent and identically distributed Bernoulli trials before $r$ failures occur. Using $r = 1$, we use the formula for the expected value of the negative binomial distribution and get the expected number of failures before total failure occurs $T_N$:

$$T_N(U) = \frac{pr}{1-p} = \frac{1-G(R_T)}{1-\left(1-G(R_T)\right)} = \frac{1-G(R_T)}{G(R_T)}$$

Note that since the number of failures in a given period time increases linearly with $k$, the expected time until total failure $T_T$ is:

$$T_T(U) = \frac{T_N \cdot F_T}{k}$$

We can now determine U by setting $T_T$ to the desired expected time till total failure, and, as in other cases, numerically find the zero of the resulting equation.

Unfortunately, for this model, the expected value is not sufficient to be useful. When $r = 1$, the negative binomial distribution has very high variance. For instance, 0 successes is frequently within 1 standard deviation of the mean, even when the mean is large.

What is frequently of more interest is the likelihood that the first failure occurs within the first $C$ successes:

$$Fail(U,C) = NegativeBinomial_{CDF}(U,C)$$

In other words, for any particular level of overprovisioning, the above function computes the likelihood that the first failure occurs before the $C$th success. We can now choose a probability of failure $fail_p$ within some period of time $C$ (e.g. .00001% within 5 years), and then numerically find the zero of:

$$F(U) = NegativeBinomial_{CDF}(U,C) - fail_p$$

Note that we must choose the minimum $U$ from analyses I and II:

$$U = \min(U_I, U_{II})$$

Other techniques don't require a similar analysis since they rely on persistent storage, which has very high data durability, to recover state.

## 4.5 Active-Active Replay

In this strategy, each time a node fails, its state must by recovered by replaying a window of data. If all other nodes fail before the first failed node recovers, then the user will experience downtime, which will be charged against the downtime budget.

The analysis presented here assumes output may be lost during downtime, similar to the first analysis in Section 4.1. A slightly more complex analysis may be done to disallow the loss of output if desired.

Using the recovery time $R_T$, and the function g(t) from Section 4.3, we can calculate, for any number of replicas, the expected time charged per failure against the SLA, called $B_T$:

$$B_T = \int_0^{R_T} (R_T - t)\cdot g(t)dt$$

Note that when using replay as a strategy, and the window size is $W_T$:

$$R_T = W_T \cdot U$$

Therefore:

$$B_T = \int_0^{R_T} (W_T \cdot U - t)\cdot g(t)dt$$

Similar to active-active periodic checkpointing, our goal is to solve for $U$ in:

$$(1 - SLA)\cdot \frac{F_T}{N_F} = B_T$$

Similarly, we find the zero for:

$$F(U) = B_T - (1 - SLA) \cdot \frac{F_T}{N_F}$$

Again, we use a numerical technique to solve for $U$. This becomes particularly critical for $N_F > 2$, where solving directly for $U$ is very challenging. Unlike the periodic checkpointing case, there is no asymptote at 1, although $f(U)$ is still monotonically increasing and guaranteed to be negative at 0. Therefore as long as we do our binary search in the range of 0 to 1, we will still find the correct answer in all cases. Note that technically, since we are allowed to miss output, very permissive SLAs allow $R_F$ greater than 1, although, in practice, we don't exploit this in our implementation of this resiliency model, placing an upper bound of 1 on $R_F$.

Note that this approach generalizes to any number of actives, although a tool like Mathematica is, in practice, needed to derive $B_T$.

Once we determine $U$, we can compute $R_F$:

$$R_F = \frac{1}{U}$$

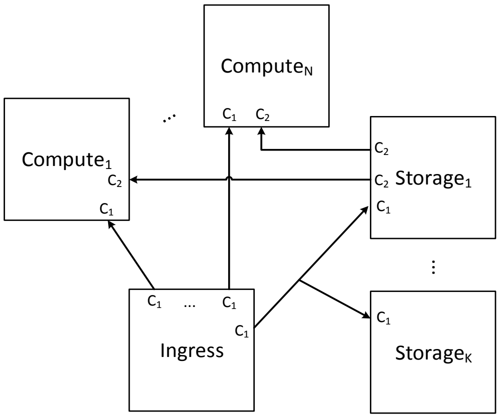

**Figure 12: Network Flow Diagram for Active-Active Replay**

To compute $C_F$, consider Figure 12, which shows the flows and costs associated with active-active with replay. First, note that cost $C_1$ is the same as in active-active with periodic checkpointing:

$$(K_F + 1 + 2 \cdot N_F) \cdot F_T.$$

The replay costs (i.e. $C_2$) are clearly similar to the costs associated with single node replay, except that replay is more common. More specifically, on average, it occurs $N_F$ times every $F_T$, incurring a cost of:

$$2 \cdot N_F \cdot (R_T \cdot R_F - R_T)$$

Combining these components allows us to conclude:

$$C_F = \frac{(K_F + 1 + 2 \cdot N_F) \cdot F_T + 2 \cdot N_F \cdot (R_T \cdot R_F - R_T)}{(K_F + 3) \cdot F_T}$$

## 4.6 Numerical Approaches

We now describe, in more detail, the numerical techniques used to find zeros of functions, needed to compute $R_F$ in many cases, and also the numerical techniques used to optimize cost for solutions which employ periodic checkpointing.

### *4.6.1 Computing $\boldsymbol{R_F}$*

All models presented in this paper, except for single node replay, compute $R_F$ by finding the zero for some $F(U)$ where $U = \frac{1}{R_F}$. More specifically, these functions have the form:

$$F(U) = C(U) - B_{SLA}$$

where $C(U)$ is the resiliency cost as a function of $U$, and $B_{SLA}$ is the allotted downtime budget for a particular $SLA$.

For all cases involving checkpointing, $C(U)$ has the desirable property that it is 0 for $U = 0$, since infinite bandwidth means no budget is actually used to checkpoint or recover, and is ∞ at $U = 1$, since without extra budget, catchup is not possible. Furthermore, it is clear that $C(U)$ increases monotonically with $U$ since more bandwidth means less resiliency cost. These properties allow us, for any particular setting of parameters, to perform a binary search for the zero in $F(U)$ without running into stability issues.

For replay based solutions, since we are not requiring that all output be produced, there is no asymptote at $U = 1$. Consider the case where the resiliency budget is so lax, that even if we replay from current input at the time the node comes up, we still have unused resiliency budget. Technically, we could, in this case, use bandwidth lower than the input rate. Note that $C(U)$ goes to ∞ as $U$ goes to infinity. So we can still perform a binary search once we find a value of $U$ s.t. $F(U) > 0$. We can then perform a binary search as in the previous case. Finding such a value isn't difficult since $C(U)$ and $F(U)$ still both monotonically increase with $U$.

### *4.6.2 Optimizing $\boldsymbol{C_F}$*

For both single node and active-active periodic checkpointing, in computing $C_F$, we need to determine the setting for checkpointing frequency (i.e. $C_T$), which optimizes $C_F$. Fortunately $C_F(C_T)$ has a straightforward shape in both cases, which allows us to approach this in a straightforward manner.

Recall that the function for $C_F$ for single node and active-active periodic checkpointing are, respectively:

$$C_F = \frac{(K_F + 3 + R_F) \cdot F_T + S_T + \frac{C_T}{2} + \frac{K_F \cdot S_T \cdot F_T}{C_T}}{(K_F + 3) \cdot F_T}$$

$$C_F = \frac{(K_F + 1 + 2 \cdot N_F) \cdot F_T + N_F \cdot 2 \cdot (S_T + \frac{C_T}{2}) + \frac{(K_F + 1) \cdot S_T \cdot F_T}{C_T}}{(K_F + 3) \cdot F_T}$$

Rewriting these functions in terms of values which depend on $C_T$, they are, respectively:

$$C_F = a_1 + a_2 R_F + a_3 C_T + \frac{a_4}{C_T}$$

$$C_F = b_1 + b_2 C_T + \frac{b_3}{C_T}$$

where $a_1 .. a_4, b_1 .. b_3$ are positive constants

Considering active-active first, it is not hard to see that this curve has a single minimum, which is approached, as $C_T$ increases, for as along as $\frac{b_3}{C_T}$ reduces faster than $b_2 C_T$ increases resulting in a graph like the one shown in Figure 13.

Such minima can be easily found using an approach similar to binary search, over a region that's known to contain the minima. One simply samples two equidistant points in the middle, and removes either the leftmost or rightmost third, ensuring that the resulting region still contains the minimum. Like binary search, this approach doesn't suffer from numerical stability issues.

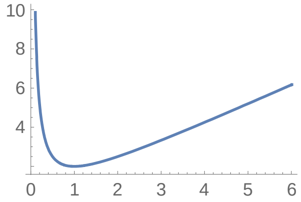

**Figure 13: Plot of $f(x) = x + \frac{1}{x}$**

For single node periodic checkpointing, the shape of $R_F$ as a function of $C_T$ is very similar to Figure 13. Initially, there is significant savings in transmitting checkpoints less frequently. Eventually, though, the added cost of replay dominates the benefit of infrequent checkpointing, leading to an optimal setting for $C_T$. $C_F$, for this case, ends up being the sum of two functions with monotonically increasing derivative, where both derivatives start negative and become positive. As a result, there can be only one point where the sum of these two derivatives is 0, where the minimum cost occurs. As a result, the overall shape of the cost function is similar to Figure 13, and the same technique may be used for optimizing cost.

# 5 MODEL VALIDATION

This section validates the accuracy of the models presented in this paper by comparing the predicted results of applying the model to actual results achieved using a distributed systems emulator that runs an actual streaming query using a real streaming data processor over real advertising data. By executing a real query with real checkpoints, these experiments also show the effect of dropping the first assumption in Section 3.2 with respect to checkpoint size, as well as dropping the third assumption of no failures during recovery. We show that our models achieve an actual SLA typically within 1% of the target.

## 5.1 The Shrink Emulator

In order to evaluate our models, we built a distributed system emulator which executes a real query over real data using the Trill streaming query processor [20]. The input to an emulator run consists of the input to our models, except for the $SLA$, as well as the output of applying our models, including the bandwidth overprovisioning factor $R_F$, and the optimized checkpointing frequency $C_T$ where appropriate.

Our system is an emulator in the sense that we have a virtual global clock which ingresses data into the streaming engine/s in accordance with an input bandwidth rate. Failures for all running copies are also randomly generated and scheduled according to an exponential distribution with the mean time to failure $F_T$. Where appropriate, actual query checkpoints are taken in Trill according to the schedule specified by $C_T$.

Upon both checkpointing and failure, $R_F$ is used to determine the length of time until normal processing resumes based on both the actual last successful checkpoint (size and virtual time), and the amount of input needed to be processed in order to catch up. The observed virtual downtime is then measured for each run, and compared to the SLA target used to generate $R_F$ and $C_T$.

In other words, we are emulating the network, and removing CPU and storage as potential bottlenecks, since the models presented so far do not consider these. Note that Section 7.1 contains a complete description of how the models presented in this paper can easily be extended to handle these other resources.

## 5.2 Experiments

These experiments consist of a series of paired model and emulator runs. For each run, parameter settings were chosen, including an uptime SLA, and a checkpoint size, which was measured in Trill using the tested query on the first part of the dataset. These parameter settings were then run through each of our models, which in turn compute $R_F$ and, in some cases, $C_T$. All of these parameters (except the uptime SLA), were then used to emulate each strategy. The actual downtime was then measured, and compared to the target SLA fed to our models.

The data were a random subset of <UserID, Search> pairs from Bing, spanning about a 2 weeks. The query was a grouped count aggregate, where the grouping field was $(UserID \ mod \ k)$, where $k$ was varied to change the size of checkpoints relative to the input that generates them. We varied the following parameters:

- Target uptime SLA – 2 nines to 5 nines
- $S_T$ (when applicable) – Varied by varying $k$, which resulted in a range of 1.4 to 611.3
- $W_T$ (when applicable) - .01 to 100
- $N_F$ (when applicable) –2 to 5

In all experiments, $F_T$ was 100. In other words, we varied the checkpoint size between ~one 100th of a failure period and ~6 times a failure period. The window size was varied between one ten thousandth of a failure period and one failure period.

Figure 14 shows the actual uptime measured by the emulator given the target uptime used to generate $R_F$. In all runs, the predicted uptime was very close to the actual runtime, with very small variations caused by randomness in failure and slight variation in checkpoint size. Note that on-demand checkpointing isn't included since all copies failed before getting a reliable value for uptime.

# 6 MODEL ANALYSIS

Through a series of parameter explorations of the models presented in Section 4, this section establishes the following about the resiliency techniques modeled in this paper:

- One size doesn't fit all: There is no single resiliency strategy which efficiently covers most of the streaming query space. Specific strategies can be vastly better or worse compared to others, depending on scenario and environment characteristics. This is true even when considering only realistic scenarios (by orders of magnitude!).
- No actionable "rules of thumb": While some strategies are better than others for specific scenarios, the tradeoffs are quite complex, and a model is needed to wade through the efficacy of different approaches for different scenarios.
- Active-active periodic checkpointing, an obvious generalization of single node checkpointing, is not discussed in the literature, likely due to the intuition that it is inferior to active-active on-demand checkpointing. Our models, however, show this strategy to be superior in many situations.

## 6.1 Computing $R_F$ and $C_F$

While our model for single node replay allows computation of our metrics directly from the parameters, all other techniques require that we numerically find the zero of a function of $U = 1/R_F$ in order to determine the value of $R_F$ which exactly consumes all available budget.

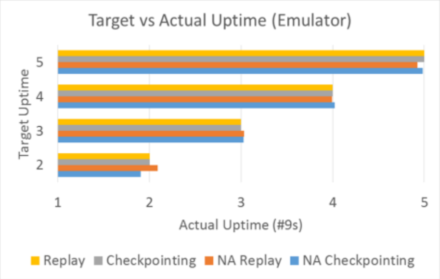

**Figure 14: Target vs Actual Uptime**

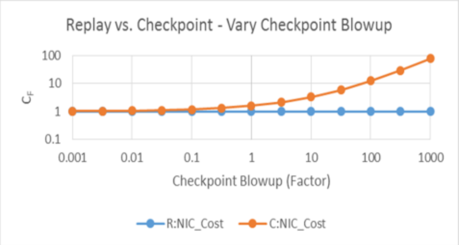

**Figure 15: Replay vs. Checkpoint - Vary Checkpoint Blowup**

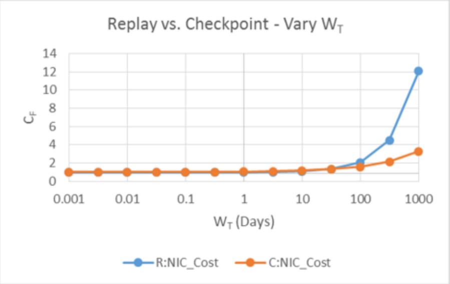

**Figure 16: Replay vs. Checkpoint - Vary $W_T$**

Recalling Section 4.6, the shape of these functions is straightforward, in that they monotonically increase with $U$ between 0 and 1, which are the bounds of interest. This allows us to do a binary search, avoiding the instabilities associated with techniques like Newton's method. Additionally, techniques that use redundant active configurations have an additional layer of complexity in the expression of these functions in that the functions vary depending on the number of actives in the configuration, and can become extremely complex (e.g. the most complex function consumes an entire screen in Visual Studio). These functions are computed automatically from the vastly simpler integrals presented in this paper using Mathematica, for specific settings of $N_F$.

Finally, the techniques in this paper which periodically checkpoint must determine the setting of $C_T$ which optimizes some notion of cost. At times we will actually optimize for minimal $C_F$. At other times we will find the optimal $C_F$ for which some bound on $R_F$ is met. For instance, we might find the setting for $C_T$ which minimizes $C_F$ where $R_F$ is at most 2. Once again, we exploit the shape of the cost curves to provide fully stable optimal solutions. In this case, the curves are more complex, as they have a single peak, so a simple binary search is insufficient. Recall that the details for numerically solving such problems are described in Section 4.6.

## 6.2 Experimental Setup

When comparing the resiliency approaches in this paper, there are generally known qualitative "rules of thumb", which state conditions under which some of these techniques will be superior. For instance, replay based solutions tend to work better when checkpoints are large compared to the input that generates them. Also, there is a general consensus that active/active solutions become more attractive as the SLA becomes more difficult to satisfy. But where exactly are the cross-over points? And how harsh is the penalty for choosing the wrong technique? Are there other factors to consider? Also, we are the first to propose active-active periodic checkpointing solutions (for streaming). How do they compare to the on-demand checkpointing based solutions?

These questions and others will be explored through a series of experiments which evaluate $C_F$ and $R_F$ for specific resiliency techniques and parameter settings using the models and evaluation techniques described in this paper. It is our intent to make available both the C# model evaluation code with which these experiments were conducted, as well as the Mathematica scripts used to integrate the functions described in our models.

## 6.3 Single Node: Replay vs. Checkpointing

We start with scenarios that contrast replay and checkpointing. Intuitively, checkpoint size, as compared to input size, would seem to provide the most interesting source of contrast. If the checkpoint is small compared to the input that generated it, this would intuitively favor checkpointing, as we trade off checkpointing costs vs. replay costs. On the other hand, if the checkpoint is much larger than the input that generated it, this would seem to favor replay. Some situations which favor checkpointing are:

- Aggregation scenarios where the internal state is a significantly reducing rollup
- "Needle in a haystack" queries, where rare events, and the events around them are analyzed.

There are also realistic situations which favor replay. For instance:

- The query logic is very complicated, involving a large number of stateful streaming operations
- The query logic contains an operation, like a cross-product, which is highly expansionary, and is followed by another non-reducing stateful operation.

There are many scenarios, covering a wide spectrum of possibilities. Where is the crossover point? How bad do things get in the extreme cases? Does the right choice depend on something other than checkpoint size? To answer these questions, we performed a sensitivity analysis, where we varied the following:

- Window size: Varied from .001 day to 1000 days (default 1)
- Checkpoint size: Varied from .001 windows of input to 1000 windows of input (default 1)
- The uptime SLA: from 50% to 99.999% (default 90%)

In addition, for all experiments, $K_F$ = 3, and $F_T$ = 1 month. In all cases, we varied one attribute and kept the other two constant, at their default values unless otherwise specified. Initially, we test our intuition about the sensitivity to checkpoint size. The result is shown in Figure 15.

As expected, as the checkpoint size increases relative to the input size, periodic checkpointing becomes more and more costly, reaching nearly 100x the cost of an unresilient solution. In contrast the cost of replay doesn't change at all as checkpoints get larger. The story is identical for $R_F$ which, for periodic checkpointing, grows to over 400x!

Next, we consider the sensitivity of these two techniques to window size, with data reducing checkpoints (checkpoint size = 0.01). Clearly both techniques become more expensive as window size increases, but which one's cost grows faster? The result of the experiment is shown in Figure 16.

While both strategies become more expensive in response to larger windows, it is clear that replay suffers more, growing to 12x the cost of an unresilient solution, while checkpointing only grows to 3.4x the cost of an unresilient solution. The story becomes even

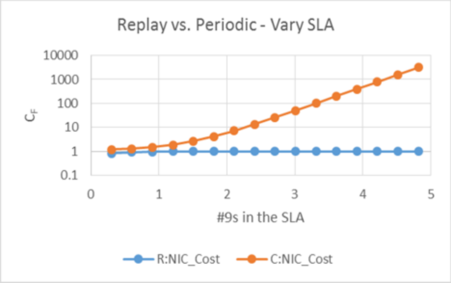

**Figure 17: Replay vs. Periodic - Vary SLA, Measure $C_F$**

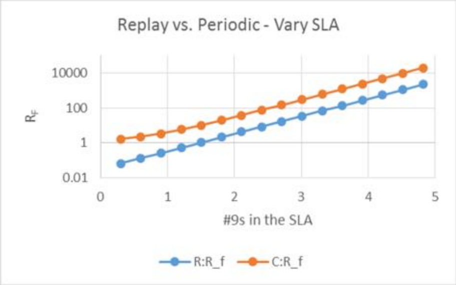

**Figure 18: Replay vs. Periodic - Vary SLA, Measure $R_F$**

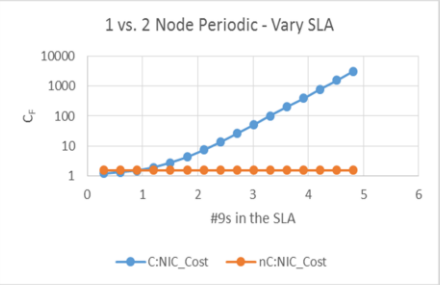

**Figure 19: 1 vs. 2 Node Periodic - Vary SLA, Measure $C_F$**

more stark when one considers the effect on $R_F$, which reaches over 300x for replay, but is only 13x for checkpointing.

The story is less intuitive when considering the effect of varying the SLA, which is shown in Figure 17 and Figure 18. Not surprisingly single node periodic checkpointing becomes highly problematic with tough SLAs, reaching costs over 3000x times the cost of an unresilient solution, and requiring network capacity on compute nodes almost 20,000x more than the input rate.

On the other hand, while replay also needs to initially find over 2000 times the input rate network capacity on compute nodes, the overall cost remains at about the cost of an unresilient solution. Once a window's worth of data is replayed on a recovering node, the cost becomes the same as an unresilient solution. The SLA only affects how quickly that window's worth of data must be replayed. In fact, for very permissive SLAs, we have longer than a window to replay the first window's worth of data, leading to costs lower than an unresilient solution, which never fails!

Checkpointing, on the other hand, continues to pay a price for tough SLAs after recovery, since the taking of each checkpoint also incurs a downtime cost, making reduction of the initial reservation untenable. It is worth noting that a tradeoff is possible with periodic checkpointing, where the bandwidth reservation is higher at the beginning, incurring a lower resiliency budget for recovery, and where that extra budget is used to lower the reservation during normal operation. This will increase $R_F$ but reduce $C_F$. We leave it to future work to examine this tradeoff.

## 6.4 Periodic Checkpointing Strategies

Conventional wisdom is that for weak SLAs, single node solutions are the most cost effective, but as the downtime SLA becomes more strict, active/active solutions become more attractive. How quickly does this effect become important, and how important? To address these questions, we compare single node periodic checkpointing with 2 node periodic checkpointing, where we vary the downtime, using the default values established in the previous section for the other parameters. The results are shown in Figure 19 and Figure 20.

First, note that the conventional wisdom concerning the cost of single vs. multinode solutions is technically correct, but practically wrong! With even just a single 9 of resiliency SLA, 2 node periodic checkpointing is already cheaper than the single node version! As the SLA becomes more strict, the advantage of using just 2 nodes becomes quite extreme. While the dominance of multinode checkpointing over single node is unintuitive at first, one must consider that with a single node, there is no spare to cover both checkpointing and recovery costs. As a result, with a single node, we must overprovision networking during normal operation to cover the times during which more bandwidth is needed for checkpointing. The multinode version doesn't suffer from this problem, which is why cost is independent of the SLA.

On the other hand, $R_F$ for multinode periodic checkpointing isn't independent of the SLA, and becomes quite high for tough SLAs with just 2 nodes. Fortunately, we can use more nodes to combat this problem. We therefore studied the effect of varying $N_F$ for tough SLAs. In this experiment, we used default values for all parameters except SLA, which was .99999, and the number of nodes, which we varied. The results are shown in Figure 21.

Increasing the number of nodes reduces $R_F$, while increasing costs. Fortunately, the cost increase isn't prohibitive, with the lines crossing at about 5 nodes, where $R_F$ and $C_F$ are both about 2.5.

## 6.5 Multinode Periodic vs On-Demand Checkpointing

In our evaluation so far, we have only considered periodic checkpointing. In fact, the literature on multinode checkpointing focuses exclusively on on-demand checkpointing. To our knowledge, we are the first to suggest that this obvious generalization of single node checkpointing could be worth considering. The purpose of this section is to, therefore, understand the networking cost of multinode periodic checkpointing as compared with on-demand checkpointing.

The comparison is complicated by the fact that while all techniques other than on-demand checkpointing are backed by highly reliable storage systems (e.g. 11 9s over a 1 year span for S3 [21]), on-demand, lacking such a stabilizer, must also meet a durability SLA (see Section 4.4). In our experiments, here, we chose the maximum $R_F$ between the two types of analysis needed to meet all SLAs (uptime and durability). We chose as our durability SLA for on-demand checkpointing, 5 9s of durability over a span of 10 years. This is actually a much weaker durability requirement than S3 provides.

The comparison is further complicated by the existence of two tunable parameters, the checkpointing period (i.e. $C_T$), which is a parameter for periodic checkpointing, and the number of replicated compute nodes (i.e., $N_F$), which is a parameter for both strategies. In addition, one can trade off $R_F$ and $C_F$ for both strategies by varying the number of nodes.

In order to compare the techniques in a sensible way, we therefore, for a particular scenario, vary $R_F$, and calculate the optimal $C_F$ across all possible settings of $C_T$ and $N_F$. This optimal value is calculated by calculating the optimal $C_F$ for each setting of $N_F$ between 2 and 10 which is guaranteed to have $R_F$ of at most the target. When reaching the target $R_F$ is not possible, that setting for $N_F$ is not used. Calculating the optimal $C_F$ for a particular setting

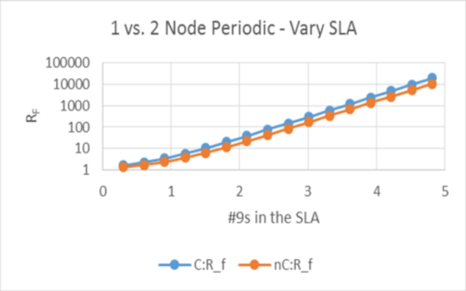

**Figure 20: 1 vs. 2 Node Periodic - Vary SLA, Measure R$_F$**

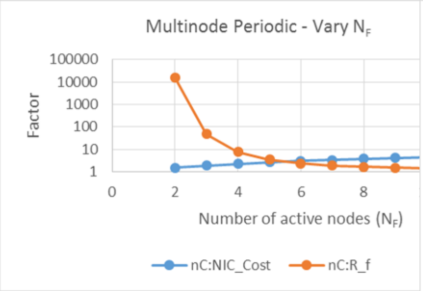

**Figure 21: Multinode Periodic - Vary N$_f$, Measure C$_F$ & R$_F$**

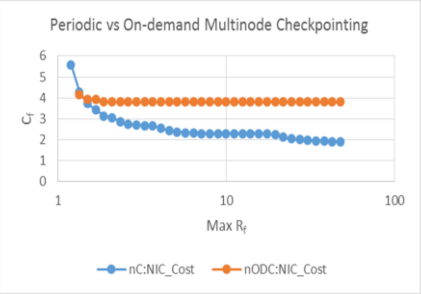

**Figure 22: Periodic vs On-Demand Checkpointing - Vary R$_f$**

of $C_T$ and $R_F$ is straightforward and is described in the Appendix. Note that we use approaches that are guaranteed to be stable, and are not approximations, as in previous calculations.

We now compare the two multinode checkpointing strategies, choosing default values for all parameters except the SLA, which is set to .99999. The results are shown in Figure 22.

First, note that for almost every case, periodic checkpointing is actually cheaper than on-demand checkpointing! The cross-over point is actually at about 2, which is not a very large value for $R_F$.

# 7 EXTENDING & APPLYING MODELS

In this section we show how our models can be extended and applied to cover a wide variety of resiliency variants and scenarios.

## 7.1 Incorporating Storage, Memory, CPU

To this point, we have only considered reservation sizes and costs for networking. We now consider other resource costs.

### *7.1.1 Storage Cost*

Storage device bandwidth is modeled in a fashion similar to network, except that we assume that modern distributed stores can perfectly load balance any storage activity, removing any local bottlenecks in the storage subsystem. As a result, there is no analog to $R_F$; only increased costs, which exactly correspond to the already calculated input logging and checkpoint reading and writing costs. These costs are then added the network costs after being multiplied with a cost normalization factor (i.e. a byte of storage bandwidth costs more than a byte of network bandwidth).

More specifically, referring back to Figure 6, the un-normalized storage bandwidth cost, $SC_F$, for single node replay, is $C_2$ plus the cost of journaling the input:

$$SC_F = \frac{R_T \cdot R_F - R_T + K_F \cdot F_T}{K_F \cdot F_T}, R_T = F_T \cdot (1 - SLA)$$

Active-active replay is very similar, except that there are $N_F$ compute nodes, each of which independently fail and recover:

$$SC_F = \frac{N_F \cdot (R_T \cdot R_F - R_T) + K_F \cdot F_T}{K_F \cdot F_T}, R_T = \frac{F_T \cdot (1 - SLA)}{N_F}$$

Considering Figure 8, storage cost for single node checkpointing is simply the edges leading to storage in the cost diagram, which is $C_2 + K_F \cdot (C_3 + C_1)$:

$$SC_F = \frac{K_F \cdot F_T + S_T + \frac{C_T}{2} + \frac{K_F \cdot S_T \cdot F_T}{C_T}}{K_F \cdot F_T}$$

For active-active periodic checkpointing, referring back to Figure 10, the storage costs are $K_F \cdot (C_1 + C_3) + N_F \cdot C_2$:

$$SC_F = \frac{N_F \cdot \left(S_T + \frac{C_T}{2}\right) + K_F \cdot F_T + \frac{K_F \cdot S_T \cdot F_T}{C_T}}{K_F \cdot F_T}$$

For active-active on-demand checkpointing, there are no storage costs beyond the already included cost of logging the input:

$$SC_F = 1$$

### *7.1.2 CPU Capacity and Cost*

For CPU cost, we introduce two new input parameters for CPU:

- $E_C$= CPU cost of taking, sending checkpoint (cost, e.g. cycles)
- $I_P$= Input processing cost (cost/time, e.g. cycles/sec)

Note that these parameters are exactly CPU centric versions of $C_S$ and $I_R$. Similarly, we introduce:

- $E_T = \frac{E_C}{I_P} =$ The checkpointing CPU time assuming input processing time CPU bandwidth (time)

Again, note that $E_T$ is exactly the CPU centric version of $S_T$. We now treat CPU like network, where CPUs can process load associated with both input processing and checkpointing. Using this approach, we introduce:

- $RCPU_F$ = The recovery CPU capacity reservation needed to meet the SLA, as a factor of input processing capacity (factor)

Note that $RCPU_F$ is exactly the CPU centric version of $R_F$. In addition, the calculation of $RCPU_F$ is identical to the calculation of $R_F$, except that $E_T$ is used instead of $S_T$. Furthermore, **both $R_F$ (for network) and $RCPU_F$ (for CPU) resource reservations must be satisfied in order for the query to meet its SLA**

In terms of cost, referring back to Figure 6, the un-normalized CPU capacity cost, $CC_F$, for single node replay, is $C_1 + C_2$:

$$CC_F = \frac{R_T \cdot RCPU_F - R_T + F_T}{F_T}, R_T = F_T \cdot (1 - SLA)$$

Active-active replay is very similar, except that there are $N_F$ compute nodes, each of which independently fail and recover:

$$CC_F = \frac{N_F \cdot (R_T \cdot RCPU_F - R_T + F_T)}{F_T}, R_T = \frac{F_T \cdot (1 - SLA)}{N_F}$$

In single node periodic checkpointing, since the CPU reservation size doesn't change, it's simply that reservation size, substituting $E_T$ for $S_T$ in the calculation:

$$CC_F = CPUR_F$$

For active-active periodic checkpointing, referring back to Figure 10, the CPU costs for this case are $N_F \cdot (C_1 + C_2) + C_3$:

$$CC_F = \frac{N_F \cdot (F_T + E_T + \frac{C_T}{2}) + \frac{E_T \cdot F_T}{C_T}}{F_T}$$

Referring to Figure 11, the CPU cost for active-active on-demand checkpointing is captured by holding the CPU reservation for the whole duration of time:

$$CC_F = N_F \cdot CR_F$$

Memory costs are easily calculated by adding up the number of reserved bytes across all nodes, which requires the addition of a parameter, and then combining with other costs after normalizing. Again, there is no analog to $R_F$ for this resource. We therefore introduce:

- $MC_F$ = The un-normalized memory cost per failure period
- $M_F$ = The cost of reserving the necessary memory per billing period compared to the NIC cost of receiving the input on a compute node for that same period.
- $CPU_F =$ CPU cost of processing the input on a single compute node in comparison to the NIC cost of receiving the input on that node.
- $S_F =$ Cost of transferring the input to a single storage device in comparison to the NIC cost of receiving the input on that node.

We can now compute total cost:

$$TotalC_F = \frac{(C_F + M_F \cdot MC_F + CPU_F \cdot CC_F + S_F \cdot SC_F)}{1 + M_F + CPU_F + S_F}$$

## 7.2 Distributed Execution

Many streaming deployments are distributed in nature. Specifically, different parts of the streaming pipeline are executed on different machines with the output of one machine feeding as input into another machine, which computes the next portion of the pipeline. In such situations, the pipeline state associated with each node must be protected from failure.

Typically, some form of reliable enough (relative to the uptime SLA) queuing is used to protect the conversation between distributed nodes, such that when a node fails, its state can be recovered through some combination of checkpointing and replay. Note that for particularly tough SLAs, transactional queues are used which are backed by replicated storage, which can meet very high availability and durability requirements [21]. Again, a choice must be made, for each pipeline running within a node, for how protection from failure is best achieved. This paper presents the models needed to make these choices in a clear, disciplined manner.

Of course, if we have two nodes, each of which have a 5 nines uptime SLA, the resulting query uptime will be less than 5 nines, since if either fails, the pipeline stalls. If our overall goal is 5 nines of uptime, we must budget downtime less than this for each node. One mildly conservative way of doing this, assuming uncorrelated failure, is to split the downtime budget between the two nodes in a way which minimizes cost. It is possible that there will be some double counting of downtime if both nodes are down at the same time, but this is a very small portion of overall downtime.

## 7.3 Sharding

Sharding is a strategy sometimes employed in streaming settings, when there is not enough memory, CPU capacity, or network available on a single node to execute the full query. To cope with this situation, the query is leveraged to produce an effective partitioning of the input, with each partition being sent to one compute shard. Depending on the computation, the results from these sharded computations may or may not be brought back together using a *combining node*. If the results are brought back together, global time order must be maintained. For the purposes of this discussion, we assume this is the case. Note that each shard, as well as the combining node, must still be resiliently protected with one of the strategies presented in this paper.

Note that there are, therefore, two analyses which must be done. One for the shards, and one for the combining node. We assume, in this analysis, that we know the total global input rate $I_R$, and total checkpoint size for the sharded computation $C_S$. In addition, assume we know all relevant information for the query running on the combining node. Note that this information is all independent of the sharding factor $S_F$.

The result of maintaining global order is that when a single shard goes down, the entire query stalls. Effectively, this decreases the mean to failure by the sharding factor, which has a cost increasing effect. On the other hand, because each shard's state, and the input that generated it, is reduced by the same fraction, recovery times after each failure are shorter. While we do not fully study these trade-offs in this paper, we demonstrate how such a study would proceed by providing the model changes necessary in single stage map/reduce style sharding, followed by combining, assuming that the ingress nodes shuffle data to the correct streaming node:

$$F_{TNew} = \frac{F_T}{S_F} \qquad I_{RNew} = \frac{I_R}{S_F} \qquad C_{SNew} = \frac{C_S}{S_F}$$

The alternate values for $F_T, I_R, and\ C_S$ are now directly used in our models to compute $R_F$. The results tell us, for each resiliency approach, given a particular sharding factor, the overprovisioning needed to achieve our SLA. Since we assume knowledge of the relevant query characteristics for the combining node, this analysis proceeds as in the distributed execution case, where the downtime is split between the combining node, and all the shards as one unit.

Note that while we use the new failure, input rate, and checkpoint size values for computing $R_F$, we use both the original and new values for computing cost. In particular, increased compute node reservations must be reflected in the cost, but storage costs are unaffected by sharding. This is a simple alteration of the formulas.

## 7.4 Caching

Recent work [13][14] exploits the potential value of caching for workloads where query state is very large, long living, partitionable, and highly inactive after an initial period of activity.

Online advertising, a problem of such high value that large distributed systems are built for the sole purpose of solving this problem, is an example of such a workload. In particular, users' browsing and ad related activity are tracked for a long period of time (e.g. a week). But most browsing sessions are, in fact, over after a short period of time (e.g. 10s of minutes), and will not contribute further to the streaming calculation.

Keeping all the session state in expensive DRAM is a poor choice for the states which are unlikely to be accessed. The problem is exacerbated for checkpointing strategies, which repeatedly checkpoint inactive states, significantly increasing the cost of resiliency.

One solution is to push the inactive states into replicated, cheap persistent storage, and only cache, in memory, the states which are still active. This significantly reduces the memory footprint of the compute nodes, which helps with both memory cost and resiliency.

MillWheel advocates using an existing key/value store for storing inactive states, but if one is running one of the active/active resiliency techniques to protect compute nodes, a better choice could be for the replicas to store their inactive states in locally attached storage. This would completely eliminate the network traffic associated with sending the states to the distributed store.

Reasoning about resiliency for these cases is straightforward, as long as we additionally know:

- The in-memory state reduction from caching.
- The required bandwidth for sending/receiving inactive states to/from storage.

In particular, the state reduction from caching is a savings applied directly to memory costs, and checkpoint sizes. The reduced checkpoint sizes are then fed into the cost model. The bandwidth for sending and receiving inactive states is used to calculate additional storage costs, as well as network costs if a distributed key/value store is used. Note that if locally attached storage is used to store inactive states, part of the recovery cost is to transmit the cached states on other nodes, similar to failure of a node in the key/value store, which must be accounted for if such a store is used.

# 8 RELATED WORK

**Streaming Resiliency** Message-passing systems have traditionally employed a wide variety of resiliency strategies, including logging, checkpointing and redundancy; see [19] for a survey. In data stream processing systems, active-active (also called active replication, active standby, or process-pairs) approaches were first proposed in Flux [5], and were adopted by several systems [10]. Timestream [8] uses checkpointing, along with leveraging query semantics to determine how much replay is needed. D-Streams [7] treats a streaming query as a sequence of micro-batch computations, with prior micro-batch state serving as checkpoints. Several systems achieve resiliency by offloading query state [13][14][15], either to resilient databases or distributed key-value stores. In this paper, we describe and/or discuss how the Shrink framework can model such resiliency techniques.

Several research papers [1][2] argue that active replication in streaming systems suffers from a high resource overhead, e.g., doubling the number of required processing nodes. In this work, we show that depending on the required SLA, active replication may in fact be the cheapest strategy by huge margins. On the other hand, Hwang et al. [11] use analysis and simulations to similarly report that active standby is superior to passive standby as it can achieve much shorter recovery time with a similar amount of overhead. Gu et al. [4] perform an empirical evaluation of the two resiliency strategies: active standby and passive standby, and report that passive standby presents a different tradeoff from active standby: longer recovery time, but 90% less overhead. These techniques provide useful intuitions for relative costs; however, unlike Shrink, they do not take the uptime SLA into account, nor do they model varying resource reservation requirements. These factors are critical for the cloud deployments of today, and lead to the completely different analysis techniques presented in this paper.

**Offline Query Resiliency** DBMSs generally provide fault-tolerance through replication [17]; however they do not provide intra-query fault-tolerance. Phoenix [18] explores resiliency for Web enterprise applications. Techniques for query suspend and resume [16] use models to choose techniques for rollback recovery in a DBMS if a long-running query fails mid-execution, which is similar to the streaming query recovery problem. Map-Reduce provides intra-query resiliency by materializing output between the map and reduce stages, and replaying these tuples on failure. Upadhyaya et al. [9] propose a cost model for the total runtime of an online (sharded) query plan over a bounded dataset in a distributed setting, across several resiliency strategies (they do not consider active standby). In contrast, we focus on modeling resiliency overheads for real-time streaming queries in the context of an overall SLA for downtime, and include active-active solutions in the space of strategies considered.

# 9 CONCLUSIONS & FUTURE WORK

This paper has introduced the first, comprehensive, cloud friendly comparison between different resiliency techniques for streaming queries. In particular, we take a resource reservation style approach, where the reservation is allowed to decrease over time. This is highly appropriate for the multi-tenant cloud environments in which these queries typically run.

In this paper, we show that specific resiliency strategies can be vastly better or worse compared to others by orders of magnitude, there are no actionable “rules of thumb”, informative models are tractable, our models are accurate (typically within 1% in practice), and can be adapted to describe many resiliency strategies, including distributed queries, sharding, and caching. We also introduce active-active periodic checkpointing, a clear generalization of single node checkpointing, and show that it is much better than on-demand caching in most situations.

We expect this work to be expanded upon in several ways. For instance, this paper focuses specifically on streaming queries, but many distributed services in the cloud face similar design choices. The uptime guarantees they provide can likely be modeled with an approach similar to what is described here. Adapting the techniques presented here to these other settings is likely very worthwhile.

# REFERENCES

[1] A. Martin, C. Fetzer, et al. Active Replication at (Almost) No Cost. In SRDS, 2011.

[2] Z. Zhang, Y. Gu, et al. A Hybrid Approach to HA in Stream Processing Systems. In ICDCS, 2010.

[3] J. H. Hwang, Y. Xing, et al. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In ICDE, 2007.

[4] Y. Gu, Z. Zhang, et al. An Empirical Study of High Availability in Stream Processing Systems. In Middleware, 2009.

[5] M. Shah, J. M. Hellerstein, E. Brewer. Highly Available, Fault-Tolerant, Parallel Dataflows. In SIGMOD, 2004.

[6] M. Balazinska et al. Fault-tolerance in the Borealis distributed stream processing system. TODS, Vol. 33, Issue 1, March 2008.

[7] M. Zaharia et al. Discretized streams: Fault-tolerant streaming computation at scale. In SOSP, 2013.

[8] Z. Qian et al. Timestream: Reliable stream computation in the cloud. In EuroSys, 2013.

[9] P. Upadhyaya et al. A latency and fault-tolerance optimizer for online parallel query plans. In SIGMOD, 2011.

[10] J-H. Hwang, U. Cetintemel, and S. Zdonik. Fast and Highly-Available Stream Processing over Wide Area Networks. In ICDE, 2008.

[11] J-H. Hwang et al. High-availability algorithms for distributed stream processing. In ICDE, 2005.

[12] G. Jacques-Silva et al. Towards automatic fault recovery in System-S. In ICAC, 2007.

[13] T. Akidau et al. MillWheel: fault-tolerant stream processing at internet scale. In VLDB, 2013.

[14] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In OSDI, 2010.

[15] J. Meehan et al. S-Store: Streaming Meets Transaction Processing. In VLDB, 2015.

[16] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. Query suspend and resume. In SIGMOD, 2007.

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

[18] D. Lomet. Dependability, abstraction, and programming. In DASFAA 2009.

[19] 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.

[20] B. Chandramouli et al. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. In PVLDB, 2014.

[21] Amazon S3. http://aws.amazon.com/s3/.

[22] K. S. Trivedi. Probability and Statistics with Reliability, Queuing and Computer Science Applications. John Wiley & Sons, 2002.