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 *service level agreements* (*SLAs*), i.e., how often query response time is impacted by failure.

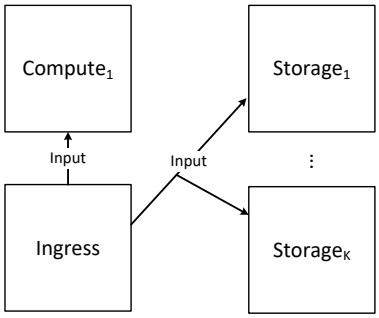

Figure 1: Typical Streaming Query Deployment

For instance, consider a typical deployment of a streaming query, shown in Figure 1. In this figure, input arrives at or is born at the ingress node. Input is then typically journaled (written) to replicated storage for later analysis, and is therefore sent to multiple storage nodes. The actual streaming computation is performed at the compute node, which may also be running other jobs. Note that compute nodes typically perform stateful computations, like windowed aggregates, which require that various counters and data structures be maintained in memory over time. As these queries are very long running, nodes eventually fail, and one of a number of proposed resiliency strategies [11] 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 compute node 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 resource 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 datacenter oriented applications.

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

Proceedings of the VLDB Endowment, Vol. 10, No. 5
Copyright 2017 VLDB Endowment 2150-8097/17/01.

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 which make the alternatives explicit and clear, and, surprisingly, only depend on a few scenario and infrastructure parameters. Our models are a major contribution, and can be applied easily without deep understanding of their derivation.
- ***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.
- ***Our models are straightforward to build upon***: Once understood, they can be adapted and extended 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***: A straightforward 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 simplest model, and in the process, introduces our modeling framework, including the intuition behind the framework, as well as our metrics and parameters, and our modeling assumptions. Section 4 then describes our most complex model, which provides a ceiling on the model complexity for the considered strategies using our framework. Section 5 describes the other resiliency strategies evaluated in this paper in enough detail to understand the experimental results. Section 6 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 7 evaluates the strategies, by applying our models with varying parameter settings. Section 8 presents the model modification for caching in systems like MillWheel. Section 9 gives an overview of related work. Section 10 concludes the paper with lessons learned and future work. Our technical report [20] presents models for the three remaining resiliency strategies, and describes the numerical approaches of Section 6.1.

2 RESILIENCY STRATEGIES

This section gives an overview of the resiliency strategies considered in this paper. We've observed that these seem to be foundational approaches, mostly described in the literature, and can be varied to create derivative solutions like MillWheel. In Section 8, we discuss some of these derivative solutions, and how the models in this paper can be adapted and applied.

These strategies are described visually in Figure 2, Figure 3, and Figure 4, which show the states of a streaming compute node (see Figure 1) 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 [20].

Note that these foundational strategies were proposed by systems like Borealis [6] and TelegraphCQ [5], which call their versions of compute nodes "processing nodes" and "query nodes" respectively.

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

Note that in all resiliency approaches described in this paper, we assume the existence of a resilient (i.e. replicated) 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.

Also, in all active-active variants, replicas are placed on different racks/failure zones, which significantly reduces correlated failure. This is typically accomplished with tools such as Azure Service Fabric [24].

2.1 Replay Based

These strategies 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 2, 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, application permitting, a point in time a bit later which minimizes catch up cost.

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 high availability scenarios, but how many copies should be run to meet an availabilty 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.

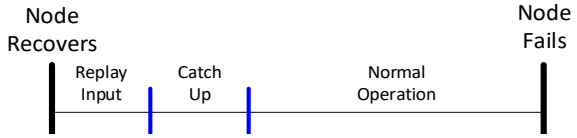

Figure 2: Replay Based Node Timeline

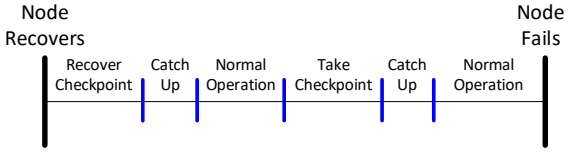

Figure 3: Periodic Checkpointing Based Timeline

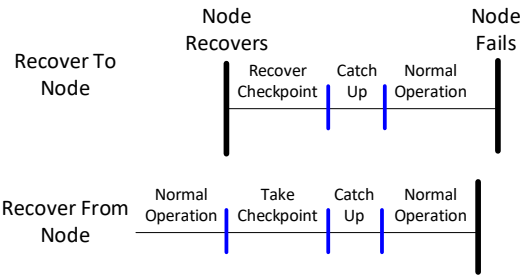

Figure 4: On Demand Checkpointing Based Timeline

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 3, 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 its 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 of recovery, checkpoints are still taken at the same points of input processing, 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 4, 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.

2.4 Resiliency Modelling Results

Strategy	Description	Evaluated
Replay All Output	Section 3	Yes
Replay Missing Output	[20]	No
AA Replay	[20]	Yes
Periodic Checkpointing	[20]	Yes
AA Periodic Checkpointing	Section 4	Yes
On-Demand Checkpointing	[20]	Yes
Including CPU and Storage	[20]	No
Caching	Section 8	No
Sharding	[20]	No
Distribution	[20]	No

Figure 5: Shrink's Current Results

This paper is the first to describe our substantial modelling effort. We now overview the Shrink project's current results, in Figure 5.

First, note that we have models for all the strategies discussed in this section, including results for both single node and active-active variants of replay and checkpointing, as well as on-demand checkpointing, which is inherently active-active.

Note, however, that due to space constraints, this paper only presents models for single node replay and active-active periodic checkpointing. Models for the other resiliency strategies may be found in our technical report [20]. Nevertheless, we include models for all these strategies in our implementation and evaluation.

Also, as discussed later, for ease of understanding, this paper focuses on models for network costs. Our technical report presents the complete models, which incorporate network, CPU, and storage costs, along with the precise model extensions needed to handle distribution and sharding. Caching extensions are presented here.

Note that these extensions are not included in our evaluation, which is designed to support our claim that even just the networking behavior of the foundational approaches is sufficiently complex to justify these models. We have, however, actually implemented the models which incorporate all resources, but feel it would unnecessarily complicate our evaluation.

3 MODELING SINGLE NODE REPLAY

In this section we present the full cost model for single node replay based resiliency. This is the simplest of our models, and is useful for establishing important modeling concepts and intuition.

3.1 Modeling Intuition

Streaming queries using replay based recovery are run on multiple nodes in a datacenter, and incur various costs, including:

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

These costs are impacted by various scenario and infrastructure parameters, and also by a **downtime SLA**. This type of SLA allows the user to specify, for instance, the maximum number of minutes per year during which the query is allowed to be "down". Down, in this context, means that the results are not being delivered in as timely of a fashion as they would if failure never occurred.

For instance, the query is down during all of recovery, since query output is delayed until the query has completely caught up to the arriving input. Note that the actual downtime experienced can be reduced by increasing the system resources for the bottlenecked resource (e.g. CPU cycles/sec, or network throughput). As a result, there is a tradeoff between allocated resources and downtime. An important and in some cases, challenging, aspect of building accurate cost models is to accurately capture this tradeoff, which is frequently necessary to make precise statements about determining overall cost.

In capturing the tradeoff between allocated resources and downtime in our cost models, we take a **resource reservation** approach. This is in contrast to previous work, which focuses on actual work done/bytes sent. Resource reservation based approaches pay the cost of reserving the resource, whether or not it is actually consumed. We use this approach because we assume that queries are run in a multitenant environment, where more than one query may be run on a single compute node. Resource capacity is then

reserved to ensure that SLAs are met, where reservations may decrease over time, but not increase, since additional capacity may not be available when needed.

In this paper, we focus on **NIC bandwidth** costs. It is useful as a proxy for overall network costs. This choice captures all network activity at the edges, regardless of internal topology, including NIC bandwidth at storage, compute, and ingress nodes. More complex models could be developed for specific datacenter network topologies, which vary widely amongst cloud providers.

Additionally, our network oriented models actually capture the mathematical complexity present in modeling other resources, and can be varied to capture the other resource costs. Our technical report [20] contains the complete extended models, incorporating all resource types into the cost models. Note that making our models sensitive to these phenomena results in a more complex, but still tractable, optimization space, which further emphasizes the need for models.

To better understand the role of NIC bandwidth reservations in single node replay, consider the network load profile of a compute node shown in Figure 6. Note that each query on a node begins its life by recovering a previously failed query's state.

Once recovery is complete, the load settles down to the bandwidth needed to receive arriving input. 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 reservation once recovery is over, leaving room on the node for other work.

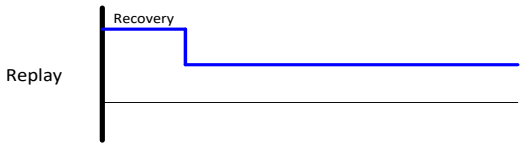

Figure 6: NIC Load for Single Node Replay

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

1. How much NIC bandwidth, compared to the input arrival bandwidth, do I need to reserve initially to recover my query while meeting my SLA?
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 facet of our modeling approach which greatly simplifies our modeling task, dramatically reduces the number of potential infrastructure and scenario parameters, and gives us a common baseline for comparing resiliency strategies. For instance, if single node replay is twice as costly as the non-resilient solution, and single node checkpointing is three times as costly as the non-resilient solution, we know that single node checkpointing is 50% more costly than single node replay.

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. While our models make this assumption mathematically, and the accuracy of our models reduces as this assumption fails, we found that in practice, with real data and queries, this assumption was not problematic (See Section 6). Also, our models can be used to perform strictly correct worst case analysis.
- 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.
- Failure detection and failover are instantaneous

3.3 Modeling Metrics and Parameters

Corresponding to the two questions posed at the end of Section 3.1, we introduce the following two metrics which will be computed for each resiliency option, given application and infrastructure parameters:

- RR_F = The recovery NIC bandwidth reservation needed to meet the SLA, as a factor of input bandwidth (factor, e.g. 2x).
- $Cost_F$ = The cost, in terms of total reserved NIC bandwidth, as a factor of the NIC costs associated with running the query non-resiliently (factor). This includes NIC bandwidth reservation costs at storage, compute, and ingress nodes.

Note that for all metrics and parameters, the subscripts represent the unit type. To compute the above metrics for single node replay, we introduce the following two scenario parameters:

- W_T = windows size, such as 10 minutes in a 10 minute trailing window (time, e.g. 10 minutes).
- 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)

3.4 Computing the baseline for $Cost_F$

Figure 7 illustrates the network flows when computation isn't made resilient to failure. The NIC costs associated with these flows form the baseline with which all other approaches are compared. Put quantitatively, the network cost of this non-resilient approach will be used in the denominator of every $Cost_F$ calculation

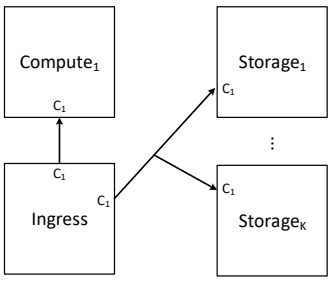

Figure 7: Network Flow Diagram With No Resiliency

From the ingress node, there is a network flow transmitting the input to the compute node, as well as 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, which must push a copy

to each of k storage nodes, but may reduce network costs with interesting topologies and/or broadcast networks. 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 model each of these copies as sent to a single node.

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 for the compute node lifetime**. For instance, if a compute node typically runs for a month before failure, assuming each of the network links shown in Figure 7 reserve just enough bandwidth to transmit the input, $C_1 = 1\ month$, since 1 month of input is transmitted over the link during that period. In other words, $C_1 = F_T$.

When computing $Cost_F$, 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.

3.5 Single Node Replay

We begin our first analysis by deriving RR_F. Note that we are trying to find the minimal setting for RR_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. For instance, if we have a $SLA = .99$, and failures happen, on average, every 100 days, then we are allowed to be down $.01 \cdot 100 = 1$ day every failure period, which becomes our recovery budget. Specifically, our budget B_T is:

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

We will now derive the time to recover, R_T, which, to exactly meet our SLA, must exactly consume our recovery budget. Since we are not allowed to miss output, recovery 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 takes 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:

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

Note that, for convenience, we will frequently substitute:

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

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:

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

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

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

In computing $Cost_F$, first consider Figure 8, 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. The total cost is straightforward to calculate, as C_2 is just the cost of reading a window's worth of data:

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

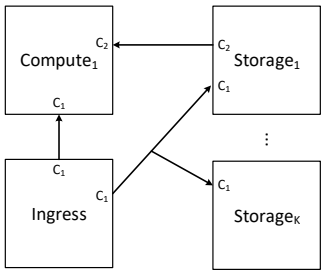

Figure 8: Network Flow Diagram for Single Node Replay

4 ACTIVE-ACTIVE PERIODIC CHECKPOINTING

We have chosen to include the model for active-active periodic checkpointing as it is the most complex model across all the resiliency strategies discussed in the paper.

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 for the first down node is charged against the SLA budget for that failure.

Similar to our discussion of single node replay, we begin our discussion by considering the network load profile of the two types of compute nodes, shown in Figure 9.

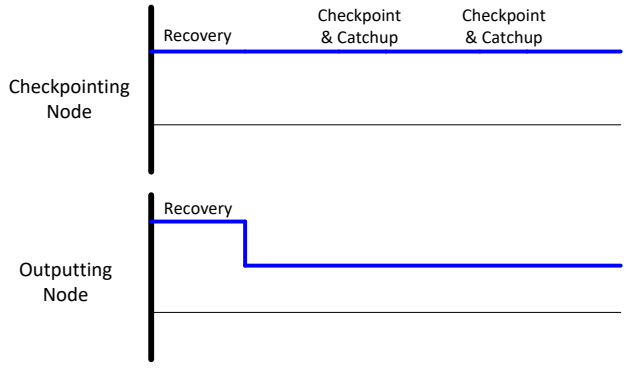

Figure 9: NIC Load for active-active periodic checkpointing

Since the checkpointing node is never used for output, it may fall behind without impacting the uptime SLA. As a result, the checkpointing node must keep up overall with arriving input, but may fall behind for periods of time. We therefore continuously

reserve the average load, rather than the peak, resulting in a constant allocation of network resources, despite periodic checkpointing. Note that nodes used for producing output will never need to produce checkpoints. These nodes have load profiles like the replay based approach, and we can similarly decrease their bandwidth reservation once recovery is over.

While both types of nodes will contribute to our cost formula, the really difficult nodes to model are the output producing nodes, where we must understand how downtime, initial network reservation size, checkpointing frequency, and the number of replicas are all related. We pursue a similar approach as the previous analysis, quantifying how much budget is consumed by initial recovery depending on initial reservation size, and use this equation to determine the initial reservation size for a particular SLA and setting for optimizable parameters. We then optimize cost by exploring the space of parameter settings and their associated costs using a straightforward hill climbing technique.

We now introduce the following additional infrastructure and application parameters for active-active periodic checkpointing:

- C_S= Checkpoint size (size, e.g. bytes)
- C_T = Checkpoint period for periodic checkpointing (time)
- I_R= Input rate (size/time)
- N_F = Number of running copies (factor).

For convenience, we also introduce 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)

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 [23]. 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.

4.1.1 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:

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

4.1.2 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 10, 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:

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

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:

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

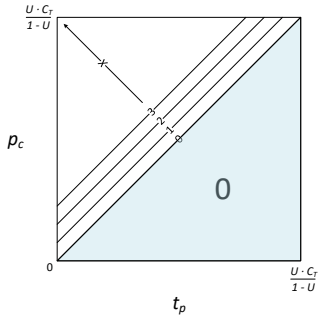

Figure 10: Case 2 variables and integration limits

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:

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

4.1.3 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. 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:

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

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, when we optimize cost, we must solve this equation for each considered setting of N_F and C_T. We will say more about how we perform this optimization from a practical point of view as part of our evaluation in Section 7.1.

Once we determine U, we can compute RR_F:

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

In computing $Cost_F$, first consider Figure 11, 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:

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

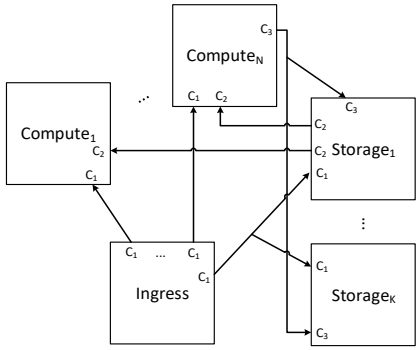

Figure 11: Network Flow Diagram for Active-Active Periodic Checkpointing

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 $Cost_F$ leads us to the following:

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

5 OTHER RESILIENCY STRATEGIES

Note that in addition to the two models presented in this paper, we have full models for the other foundational resiliency approaches mentioned earlier, and include an implementation of these models in our evaluation. The detailed models are available in our tech report [20]. In particular, this includes models for:

- Active-Active Replay
- Single Node Periodic Checkpointing
- On-Demand Checkpointing

For the purposes of interpreting the evaluation presented in this paper, we now present the network load profiles for these strategies, and discuss their implications. In particular, these profiles are shown in Figure 12.

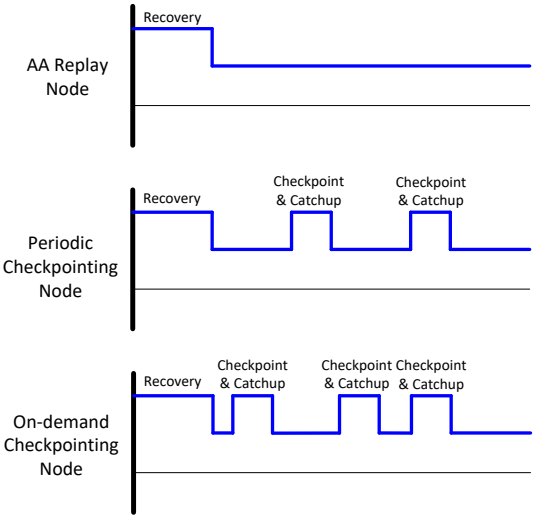

Figure 12: NIC Loads for remaining strategies

Similar to single node replay, in active-active replay, once recovery is complete, the load settles down to the load needed to process the input. Note that we must find enough bandwidth on the node to recover quickly enough to meet our SLA given the possibility of all running copies failing, similar to active-active periodic checkpointing.

For single node periodic checkpointing, there is one node responsible for both taking checkpoints, and producing output, resulting in a load profile which both increases and decreases. Note that downtime is now experienced both during recovery, and also while checkpointing and catching up. Since we are not allowed to increase the bandwidth reservation, this leads to a continuous reservation level high enough to ensure that our downtime SLA is not violated. As a result, a significant portion of our reserved bandwidth may be unused, resulting in significant resource waste. This is reflected in our cost model for this approach.

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 similar potential for resource waste.

6 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.

6.1 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 [21]. 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 RR_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, RR_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 RR_F and C_T.

In other words, we are emulating the network, and removing CPU and storage as potential bottlenecks, since these models are sufficiently complex to support our claims. Our technical report [20] contains a complete description of how the models presented in this paper can easily be extended to handle these other resources.

6.2 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 RR_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 $(UserID \bmod k)$, where k was varied to change the size of checkpoints relative to the input that generates them. We varied the following parameters:

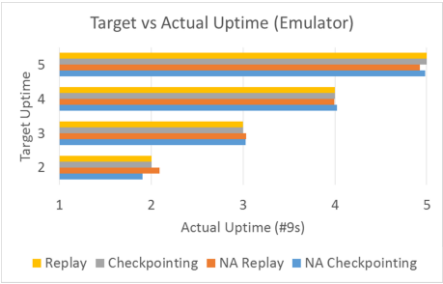

Figure 13: Target vs Actual Uptime

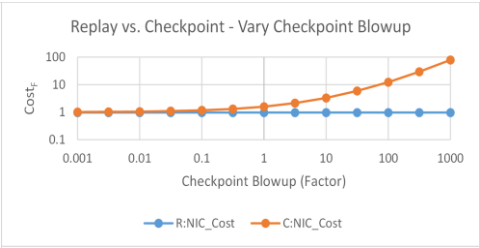

Figure 14: Replay vs. Checkpoint - Vary Checkpoint Blowup

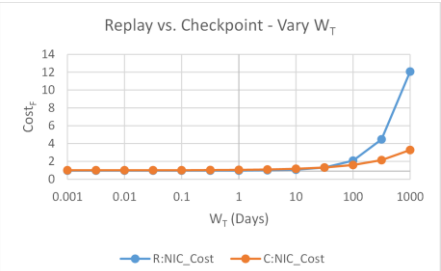

Figure 15: Replay vs. Checkpoint - Vary W_T

- 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 13 shows the actual uptime measured by the emulator given the target uptime used to generate RR_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.

7 MODEL ANALYSIS

Through a series of parameter explorations of the models presented in the earlier sections, 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.

7.1 Computing RR_F and $Cost_F$

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 $U = 1/RR_F$ in order to determine the value of RR_F which exactly consumes all available budget.

Fortunately, 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 $Cost_F$. At other times we will find the optimal $Cost_F$ for which some bound on RR_F is met. For instance, we might find the setting for C_T which minimizes $Cost_F$ where RR_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. Our numerical approaches are fully described in our technical report [20].

7.2 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 $Cost_F$ and RR_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.

7.3 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:

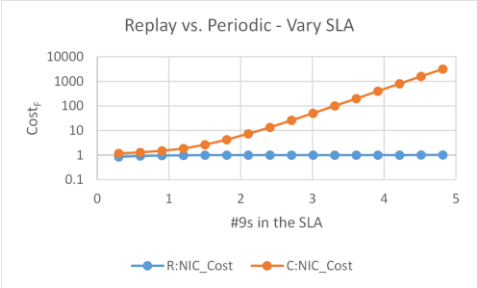

Figure 16: Replay vs. Periodic - Vary SLA, Measure $Cost_F$

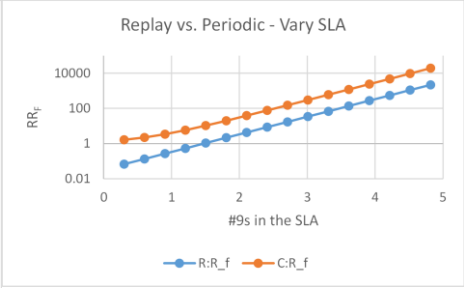

Figure 17: Replay vs. Periodic - Vary SLA, Measure RR_F

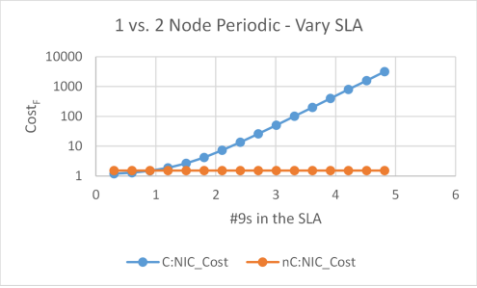

Figure 18: 1 vs. 2 Node Periodic - Vary SLA, Measure $Cost_F$

- 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 14.

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 RR_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 15.

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 more stark when one considers the effect on RR_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 16 and Figure 17. 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 RR_F but reduce $Cost_F$. We leave it to future work to examine this tradeoff.

7.4 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 18 and Figure 19.

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, RR_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

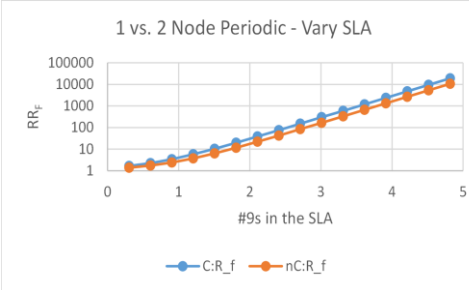

Figure 19: 1 vs. 2 Node Periodic - Vary SLA, Measure RR_F

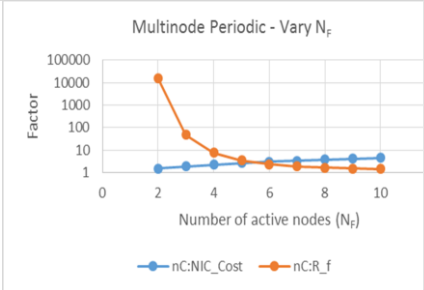

Figure 20: Multinode Periodic - Vary N_f, Measure $Cost_F$ & RR_F

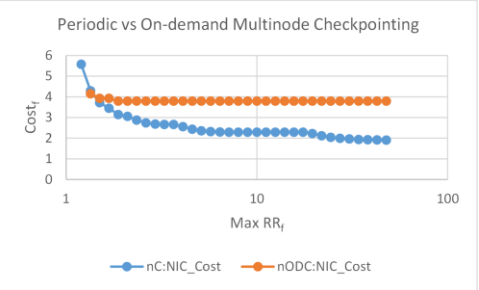

Figure 21: Periodic vs On-Demand Checkpointing - Vary RR_f

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 20.

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

7.5 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 [22]), on-demand, lacking such a stabilizer, must also meet a durability SLA. We include such a modeling approach in our technical report [20]. In our experiments, here, we chose the maximum RR_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 RR_F and $Cost_F$ 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 RR_F, and calculate the optimal $Cost_F$ across all possible settings of C_T and N_F. This optimal value is calculated by calculating the optimal $Cost_F$ for each setting of N_F between 2 and 10 which is guaranteed to have RR_F of at most the target. When reaching the target RR_F is not possible, that setting for N_F is not used. Calculating the optimal $Cost_F$ for a particular setting of C_T and RR_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 21.

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 RR_F.

8 MODELS FOR CACHING

Recent work, such as MillWheel [13], exploits the 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.

9 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 with on-demand checkpointing (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.

10 CONCLUSIONS & FUTURE WORK

This paper has introduced the first, comprehensive, cloud friendly comparison between different resiliency techniques for streaming queries. In particular, we take an uptime SLA and resource reservation driven 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.

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 and J. Goldstein. Shrink: Prescribing Resiliency Solutions for Streaming. Technical Report, Microsoft Research. http://aka.ms/shrink-tr.

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

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

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

[24] Azure Service Fabric. http://aka.ms/vv5909.