

**Scalable Fault Tolerance for High-Performance Streaming  
Dataflow**

by

Gina Yuan

B.S., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer  
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2019

© Massachusetts Institute of Technology 2019. All rights reserved.

Author .....  
Department of Electrical Engineering and Computer Science  
September 3, 2019

Certified by .....  
Robert T. Morris  
Professor of Electrical Engineering and Computer Science  
Thesis Supervisor

Accepted by .....  
Katrina LaCurts  
Chair, Master of Engineering Thesis Committee



# Scalable Fault Tolerance for High-Performance Streaming Dataflow

by

Gina Yuan

Submitted to the Department of Electrical Engineering and Computer Science  
on September 3, 2019, in partial fulfillment of the  
requirements for the degree of  
Master of Engineering in Computer Science and Engineering

## Abstract

Streaming dataflow systems offer an appealing alternative to classic MySQL / memcached web backend stacks. But websites must not go down, and current fault tolerance techniques for dataflow systems either come with long downtimes during recovery, or fail to scale to large deployments due to the overhead of global coordination. For example, in the failure of a single dataflow node, existing lineage-based techniques take a long time to recompute all lost and downstream state, while checkpointing techniques require costly global coordination for rollback recovery.

This thesis presents a causal logging approach to fault tolerance that rolls back and replays the execution of only the failed node, without any global coordination. The key to knowing how to replay a valid execution while ensuring exactly-once semantics is a small, constant-size *tree clock* piggybacked onto each message, incurring runtime overheads that are low and scalable. After recovery, the state of the system is indistinguishable from one that never failed at all.

We implement and evaluate the protocol on Noria, a streaming dataflow backend for read-heavy web applications. Compared to Noria's original protocol of lineage-based recovery, tree clock recovery time is constant in relation to state size and graph size. Experimental results show sub-second recovery times with 1.5ms runtime overheads, which translates to a 290x improvement in recovery time.

Thesis Supervisor: Robert T. Morris

Title: Professor of Electrical Engineering and Computer Science



## Acknowledgments

Robert, for his wise and deliberate advice, calming conversations, and providing direction exactly when I needed it.

Malte, for his bountiful optimism and energetic approach to research, and inspiring me to look at problems a different way.

Jon, for every time I walked up to his desk when research got tough, and left with my spirit refueled and passion renewed.

Raul, Eddie, Frans, Albert, Justin, Sam, and everyone else in PDOS and the Database Group who has shaped me into the researcher and software engineer I am today.

MIT Sport Taekwondo, especially Yang, Rachel, Renee, Andrew, Tahin, and Master Chuang, for giving me a second home at MIT.

My parents for their endless love and support.



# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	Summary . . . . .	13
1.2	Contributions . . . . .	13
<b>2</b>	<b>Background and Related Work</b>	<b>15</b>
2.1	Noria: the dataflow model . . . . .	15
2.2	Lineage-based recovery . . . . .	16
2.3	Checkpointing . . . . .	16
2.4	Causal Logging . . . . .	17
<b>3</b>	<b>Design</b>	<b>19</b>
3.1	Normal operation . . . . .	19
3.1.1	Tree clocks . . . . .	20
3.1.2	Payload log . . . . .	21
3.1.3	Diff log . . . . .	21
3.1.4	Message processing algorithm . . . . .	22
3.2	Failure . . . . .	23
3.2.1	Controller . . . . .	23
3.2.2	Recovery protocol . . . . .	24
3.2.3	Deduplication . . . . .	26
3.3	Ensuring a valid execution order . . . . .	26
3.3.1	Execution replay table . . . . .	27
3.3.2	Solving the ER table . . . . .	28

3.3.3	Using the ER table to replay messages . . . . .	30
3.4	Optimizations . . . . .	31
3.4.1	Neighborhoods . . . . .	31
3.4.2	Log truncation . . . . .	32
<b>4</b>	<b>Implementation</b>	<b>33</b>
<b>5</b>	<b>Evaluation</b>	<b>35</b>
5.1	Benchmarking methodology . . . . .	35
5.1.1	Dataflow graphs . . . . .	35
5.1.2	Hardware . . . . .	37
5.2	Read staleness . . . . .	37
5.3	Read and write latency . . . . .	40
5.4	Recovery time . . . . .	41
<b>6</b>	<b>Correctness</b>	<b>45</b>
6.1	Invariants . . . . .	45
6.2	Proof: the recovery protocol produces a valid execution order. . . . .	46
6.2.1	Each $C_i$ 's inputs reflect a valid execution after recovery. . . . .	46
6.2.2	$B$ 's inputs reflect a valid execution after recovery. . . . .	47
6.2.3	$B$ 's outputs determine valid inputs to each $C_i$ after recovery. . . . .	48
6.2.4	The recovery protocol is correct with log truncation. . . . .	48
6.2.5	Summary . . . . .	49
6.3	Proof: eventually, the system can recover again after a failure. . . . .	49
<b>7</b>	<b>Extensions and Future Work</b>	<b>51</b>
7.1	Stateful recovery . . . . .	51
7.2	Multiple concurrent failures . . . . .	51
7.3	Other work . . . . .	52
<b>8</b>	<b>Conclusion</b>	<b>53</b>

# List of Figures

2-1	Processing a write in the dataflow graph. . . . .	16
3-1	Bookkeeping state in a node for recovery. . . . .	20
3-2	A node's tree clock. . . . .	20
3-3	A node's payload log. . . . .	21
3-4	A node's diff log. . . . .	22
3-5	A node receives a message and applies a diff to its tree clock. . . . .	22
3-6	A failed node with multiple parents and multiple children. . . . .	23
3-7	Recovery protocol officiated by the controller. . . . .	25
3-8	An invalid execution as the result of processing messages in the order they were received. . . . .	26
3-9	Table used to determine a valid execution replay. . . . .	27
3-10	Empty ER table for the restarted node $B'$ in Fig. 3-6. . . . .	28
3-11	Algebraic approach to solving the ER table. . . . .	29
3-12	The diff log of the restarted node after recovery. . . . .	31
3-13	The apply diff operation with tree clocks of bounded size. . . . .	31
4-1	The clock data structures in the implementation . . . . .	33
4-2	Recovery-related bookkeeping for each node in the implementation. . . . .	34
4-3	The messages exchanged in the recovery protocol, in the implementation. . . . .	34
5-1	The SQL tables and queries used in the benchmark. . . . .	36
5-2	The sharded dataflow graph used in the performance benchmark. . . . .	37
5-3	Offered load vs. write propagation time. . . . .	38

5-4	The size of message components leaving the sharder node. . . . .	39
5-5	Offered load vs. the number of ways a sharder splits an outgoing message. .	39
5-6	Offered load vs. read latency . . . . .	40
5-7	Offered load vs. write latency. . . . .	40
5-8	Write propagation over time for each recovery protocol. . . . .	41
5-9	Number of articles vs. recovery time. . . . .	42

# Chapter 1

## Introduction

Noria is a streaming dataflow system for read-heavy web application backends, intended to replace the classic MySQL / memcached stack [16]. In this use case, Noria requires the resources of many computers to support thousands of latency-sensitive end users simultaneously accessing hundreds of materialized views. Noria shards the backend to spread the work across machines, a common approach to distributing load [1]. This results in dataflow graphs that may have hundreds of nodes spread over multiple computers.

At the scale of a large website, which may necessitate hundreds or thousands of shards to keep up with load, machine failures are inevitable. When failures happen, the backend must be able to restore the system to a *globally-consistent* state, which is a state the system could have been in had no failure happened at all. Like many other database and dataflow systems [14, 20], Noria maintains a strict set of correctness guarantees across its multiple machines. Noria requires materialized views to be eventually-consistent, and each node must process updates exactly-once and in the same order they were received. Meanwhile, the system must remain online to not disrupt users' abilities to use the web application.

Consider the failure of a single computer in the fail-stop model of failure [23]. The computer immediately loses the dataflow nodes on the computer, messages in transit to and from the computer, and any materialized state in the nodes. For simplicity, assume the failed computer has a single dataflow node, and that the node does not have any materialized state.

Imagine we restarted the failed node on a new computer. If upstream nodes continued as normal, downstream nodes would never receive the messages lost in the failure. We can replay lost messages from the in-memory logs of the restarted node's parents, but then we would need to know exactly where to start to avoid sending duplicates or losing a message. Even if we knew where to start, the restarted node may interleave messages from its parents in a different order, producing an output order that is inconsistent with messages some of its children have already seen. The non-determinism in execution order after failure is a well-established problem in similar approaches [13, 4, 28].

To avoid the complexity of tracking where and what order messages were sent, Noria currently purges the node's entire downstream graph and recomputes the state from scratch. Some nodes may have been on surviving computers, but Noria redundantly purges and recomputes their states as well. Like in other coarse-grained lineage recovery solutions [30, 28], recovery time with this protocol is proportional to the size of state in the

graph. Rebuilding the state can take days or even weeks with large amounts of historic data. This is a problem for long-running web applications with many clients, where massive amounts of data accumulate over time and high availability is mission-critical [15, 17].

An alternative is to restore the graph from a previously-consistent checkpoint without having to rebuild the state from scratch. Global checkpointing solutions incur high runtime overheads due to the need for global coordination in the normal case [22, 21]. Distributed checkpointing eliminates overheads in the normal case [6, 8], but still requires costly global coordination to roll back the entire graph in the event of the failure. In this case, the recovery time is proportional to the number of dataflow nodes in the graph. As a result, checkpointing also does not scale to the large, complex graphs we expect from Noria.

Causal logging is a class of fault tolerance techniques that rolls back only the failed node to return the system to a globally-consistent state [4, 13]. The main idea behind causal logging is that a valid execution only needs to observe the causal effects of previous messages, not to be exactly the same. Some causal information, or data lineage, is piggybacked onto each existing message and tracked in each node. During recovery of a failed node, the remaining nodes piece together their causal information to produce a valid execution. In particular, they use the lineage to determine which messages to replay and what order to process them in to reflect a valid interleaving of messages.

The naive approach to causal logging incurs large runtime overheads. Since the lineage information is passed through the data plane, every additional message compounds the overhead of piggybacked lineage information [4]. While batch processing systems limit the frequency of messages [18, 12, 31, 7], continuous stream processing systems may send millions of messages per second at milliseconds latency [22, 19, 3]. The lineage information itself can also be large and proportional to the size of the graph. As a result, causal logging techniques traditionally trade-off the accuracy of lineage for lower runtime overheads [4].

This thesis presents a new causal-logging approach to fault tolerance for Noria. This solution incurs low runtime overheads while guaranteeing exactly-once semantics, against the assumption that consistency with causal logging requires a heavy runtime cost. The key to achieving low runtime overheads in Noria is a small, constant-size *tree clock* that includes only the *changed* lineage information, and whose size is independent of the structure of the graph. By piggybacking the constant-size tree clock onto each message, we encapsulate enough information to restore the system to a globally-consistent state, without any global coordination. Our current implementation does not explicitly support concurrent failures of multiple nodes or failures of stateful nodes, but we believe that our model can easily be generalized to these use cases in future work.

In exchange for forwarding only small amounts of lineage information, our solution requires a more complex *recovery protocol*. A controller officiates the recovery protocol in the coordination plane, as opposed to the data plane, and invokes a series of message exchanges between the controller, the restarted node, and the immediate neighbors of the restarted node. If the node’s execution is necessarily deterministic, the controller only needs to determine where each node should resume sending messages from its log. Otherwise, the controller invokes an *execution replay* algorithm to determine the interleaving of messages received by the restarted node.

## 1.1 Summary

In summary, this thesis presents a scalable approach to fault tolerance using a low-overhead causal logging technique called *tree clocks*, combined with a *recovery protocol* for deterministic failures and an *execution replay* algorithm for non-deterministic ones. The approach is scalable because we can add more machines to the system without impacting recovery time or runtime overhead. We implement the approach on Noria with the following correctness and performance goals:

- **Correctness:** Updates are reflected in the outputs exactly-once and are processed by nodes in the order they are sent. If no new updates are made, materialized views eventually reflect all updates up to the last updated value i.e. eventual consistency.
- **Performance:** Recovery time and runtime overhead are constant in relation to the amount of state and the number of nodes in the dataflow graph. Both are reasonably low.

Currently, the scope of our model only includes computer failures that affect a single dataflow node, which must be stateless. The model also does not currently support partially-stateful and dynamic dataflow graphs. However, we believe the recovery protocol is general enough to integrate these properties, which we address in Future Work (§7).

## 1.2 Contributions

The contributions of this thesis are:

- **Tree clocks:** an abstraction for tracking data lineage, which is where and in what order messages are sent, with low-overhead local coordination.
- **Recovery protocol:** a scalable fault tolerance protocol that uses the causal logging information in tree clocks to recover from deterministic single-node failures.
- **Execution replay:** an algorithm that combines with the recovery protocol to resolve non-determinism in failures and produce a globally-consistent state.
- **Implementation:** a prototype in Noria with an evaluation of runtime overheads in the normal case and recovery time after failure.



# Chapter 2

## Background and Related Work

### 2.1 Noria: the dataflow model

Noria is a streaming dataflow system intended to replace the classic MySQL / memcached stack. The core component of the system is a high-performance dataflow graph layered over a traditional database. The client first specifies a commonly-used SQL query, like one that is called when loading a website frontend. Noria translates the SQL query into a dataflow graph of nodes, or relational operators, where nodes are sometimes shared by multiple queries to reuse state. The output nodes of the graph represent materialized views, and they cache pre-computed values that make reads to the query blazingly fast.

Once the graph has been initialized, clients interact with Noria by sending requests to the inputs or outputs of the dataflow graph (Fig. 2-1). Writes to Noria are first persisted to a base table, then injected into an input node of the graph. The input node then forwards the write message to downstream operators along graph edges until the message is materialized in an output node. While the write is still flowing through the graph, reads to the materialized view return a cached, potentially stale, value. Once the write has fully propagated through the graph, the reads will return an updated value.

When a node receives a message, it produces an output message as a deterministic function of the input message and any state in the node. The function also determines which children, if any, the node sends the output message to. Output messages are necessarily the result of an input message. Writes to Noria are batched, meaning a single message could potentially reflect many writes to a base table.

The sharder node is a particularly interesting case of a stateless node. In general, Noria shards a graph by replicating the graph by the system's sharding factor. If portions of the graph are sharded by different keys, which is often the case in more complex queries, Noria uses a *sharder* node to re-route the graph. The inputs to the sharder node are replicas of the upstream graph sharded by one key, while the outputs of the sharder node are replicas of the downstream graph sharded by another key. There are no inherent constraints on which nodes, if any, the resulting output of a sharder node can go to. In presenting the general case of the algorithm when losing a stateless node with multiple parents and multiple children, we have the sharder node in mind.

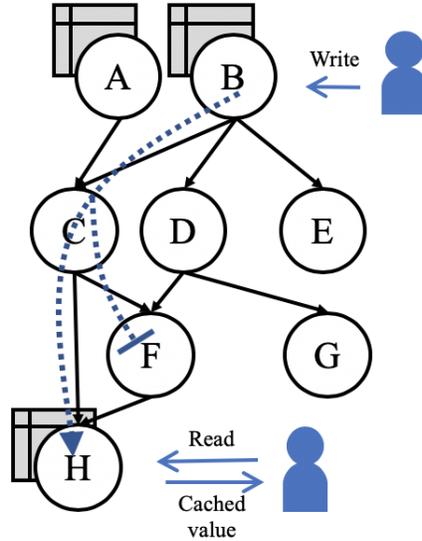


Figure 2-1: A generic dataflow graph in Noria without sharding, where  $A$ ,  $B$ , and  $H$  have materialized state. A user injects a write to  $B$ , who forwards the message just to  $C$ .  $C$  forwards the message to both  $F$  and  $H$ . While  $F$  filters the message and doesn't send it to  $H$ ,  $H$  materializes the message from  $C$ . Another user reads the materialized view in the output node  $H$  and observes the propagated write.

## 2.2 Lineage-based recovery

Lineage-based recovery is a common fault tolerance technique for bulk-synchronous parallel (BSP) systems [31, 7]. When the lineage of a message is known before processing, these systems can rebuild lost state from partial results [31, 12, 7]. But while BSP systems have natural barriers for resuming computation due to their synchronous model of execution [27, 12, 18], it is unclear where continuous stream processing systems like Noria can rebuild their state except from scratch [16]. Recomputing state can take a long time, causing harmfully long downtimes for Noria's latency-sensitive users.

## 2.3 Checkpointing

Checkpointing rolls back the system to a globally-consistent state after failure. Global checkpointing is intuitively correct, but adds high runtime overheads due to the global coordination required for each checkpoint [22, 21]. Distributed and asynchronous checkpointing eliminate the need for coordination in the normal case [5, 9], but do not necessarily guarantee exactly-once since the execution order between the time of failure and the time of the rollback may be different. To provide stronger consistency guarantees like exactly-once, the system must roll back its entire graph [5, 8], which has been shown to be slow at scale [27]. Our fault tolerance solution draws from the distributed progress-tracking ideas of distributed checkpointing, while avoiding global coordination during recovery.

## 2.4 Causal Logging

Causal logging protocols send lineage information with each message in the data plane [4, 13]. On failure, the information on surviving nodes can be used to restore the system to a globally-consistent state. Depending on the size of lineage information, causal logging may incur high runtime overheads. Lineage Stash removes the overhead from the data path by asynchronously logging lineage information to a decentralized store [28]. We present a different approach to causal logging, decreasing the size of the lineage required so that even synchronous processing of lineage incurs little runtime overhead. Also unlike Lineage Stash, our solution guarantees sequential consistency, i.e. messages from the same node are processed in the same order by its children, an important property for avoiding subtle bugs with non-determinism.

There has been much work in improving the representation of lineage to be memory-efficient. Vector clocks, whose sizes are proportional to the number of processes in the naive approach, have been made more efficient by only forwarding the part of the clock that has changed [25, 10]. Lineage Stash forwards only the most recent part of the lineage that has not been durably stored [28]. Noria potentially faces a similar memory blowup problem with tree clocks. We solve this problem by constraining the size of tree clocks based on which nodes in the graph are allowed to communicate with each other and the paths the messages are allowed to take.



# Chapter 3

## Design

We approach the problem of knowing where and in what order to resend messages by looking at the failure of a computer with a single stateless node (Fig. 3-6). Throughout this section, we call this failed node  $B$ , where  $B$  has multiple parents  $\{A_i, i \in [1, m]\}$  and multiple children  $\{C_i, i \in [1, n]\}$ .

We motivate this design by the desire to not redundantly purge and rebuild the state in nodes on surviving machines. Consider what would happen in Fig. 3-6 if we used Noria's naive approach to fault tolerance. In this approach, Noria would purge all the state in  $B$ 's children, incurring long recovery times if the nodes had to recompute a lot of state. If each of  $B$ 's children had descendants of their own, we would have to purge their state as well.

Instead, our fault tolerance solution proposes we restart the failed node on a new machine and leave the surrounding nodes intact. Call this restarted node  $B'$ .  $B'$  does not initially have any state of its own, but our solution reconstructs any relevant information in the node by using the state on surviving machines. Though we present the solution in the context of a single stateless node, we believe this approach is important for leading us towards a complete fault tolerance solution including stateful operators.

First, we present a data structure called *tree clocks* for tracking lineage, and describe and how it is used in normal operation (§3.1.1). Next, we introduce a general recovery protocol that uses tree clocks to calculate where (§3.2) and in what order (§3.3) to resume sending messages. Throughout the description of the recovery protocol, we supplement the text with an example execution. Finally, we discuss optimizations that reduce the space overhead of tree clocks and logs for practical use (§3.4).

### 3.1 Normal operation

In this section, we discuss what information a node needs to track to be able to recover the system to a globally-consistent state after failure. This recovery-related bookkeeping involves three main components: a *tree clock* for accumulating the lineage of all received messages, a *diff log* for storing changes in lineage, and a *payload log* for storing the history of data in each outgoing message (Fig. 3-1).

Field	Description
Tree Clock	Accumulate the lineage of all received messages to remember the last time the node heard from each upstream node.
Payload log	Store the data in each outgoing message to be able to replay messages to downstream nodes from a previous point in time.
Diff log	Store changes in lineage to capture the non-determinism in the order that upstream nodes received messages.

Figure 3-1: Bookkeeping state in a node for recovery.

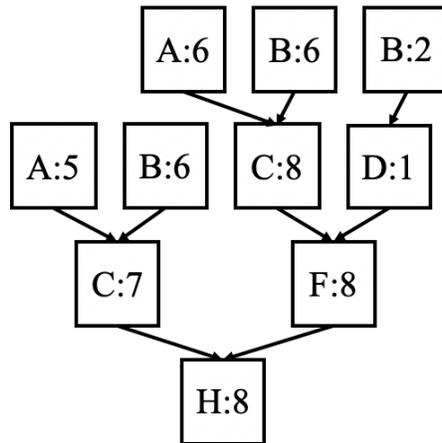


Figure 3-2:  $H$ 's tree clock based on the dataflow graph in Fig. 2-1, with root  $H$  and time  $t_H$ . Note that  $C$  appears in  $H$ 's tree clock twice because there are two paths a message can take to reach  $H$  from  $C$ . The times associated with each  $C$  are different, due to either the non-determinism with which messages arrive or due to which children  $C$  sent each message to.

### 3.1.1 Tree clocks

A tree clock is an inverted tree of node IDs. The structure of the tree clock depends on the root of the tree clock and the structure of the corresponding dataflow graph. Each node  $H$  in the dataflow graph keeps a tree clock with root  $H$  and every possible path to  $H$  (Fig. 3-2). A path in the tree clock depicts a path a message could have taken through the dataflow graph to reach  $H$ . If there are multiple paths a message can take from a certain node, the node appears in the tree clock multiple times.

Each node in the tree clock is associated with an integer *time*. This time represents a counter of outgoing messages sent by that node. However, because a node may not send all its messages to all its children, the counter does not necessarily determine the number of messages received from a node. Also, because we increment the counter for each outgoing message, we can associate each message with a unique node ID and time.

*Diffs* are a subset of tree clocks that represent the lineage of messages through the dataflow graph. Sometimes referred to as provenance [11], lineage tells us the history

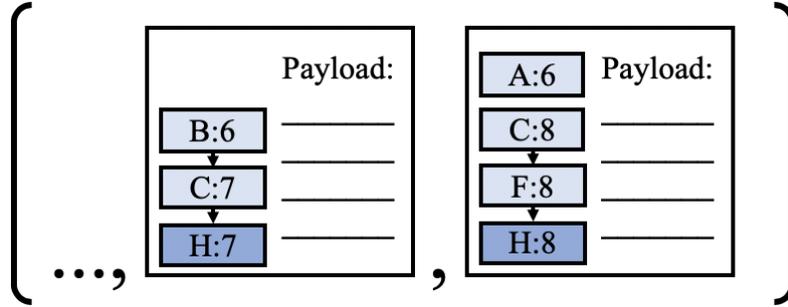


Figure 3-3: The payload log of node  $H$ , which contains all of  $H$ 's outgoing messages. Each message has a diff with root  $H$ , where  $t_H$  increases sequentially across the log. Each message also has a data payload.

of nodes a message passes through to get to where it is. In stateless nodes, where the outputs of messages do not depend on the results of previous messages, diffs are necessarily linear. Since messages are processed exactly-once, the diff associated with a message is necessarily unique.

We use the following notation to discuss tree clocks in textual form. Capital letters indicate nodes in a dataflow graph, except the letter  $T$ , which indicates a tree clock. Let  $T$  be the tree clock in Fig. 3-2. In  $T$ ,  $H:8$  refers to node  $H$  associated with time 8. Square brackets indicate an unordered layer of nodes in the tree clock. The complete textual notation for  $T$  is:

$$H:8 [C:7 [A:5 B:6] F:8 [C:8 D:1]].$$

Additionally, denote  $t_{[H,F,D]}$  to be the time associated with  $D$  in the tree clock by taking the path through  $H$  and  $F$ . When the path to  $D$  is unambiguous, we can abbreviate the notation to  $t_D$ . In this case,  $t_{[H,F,D]} = t_D = 1$ .

### 3.1.2 Payload log

The payload log is a log of all messages a node has sent. Each message contains two parts: the diff and the payload. Consider node  $H$ 's payload log (Fig. 3-3). The diff in each message tells receiving nodes that  $H$  sent the message and at which time. Since  $H$  assigns times sequentially as it produces outgoing messages, the root times of the messages also increase sequentially across the log. The payload is the actual message data that is processed and transformed in each node. In general, the payload log is necessary to be able to replay messages to downstream nodes from a previous point in time.

### 3.1.3 Diff log

The diff log is a log of the changes in lineage for each message the node has sent. The node constructs a *diff* from an input message, then sends the diff with an output message. Consider node  $H$ 's diff log (Fig. 3-4). Each diff has root  $H$ , and the root times increase sequentially across the log in the order they were assigned. The parent node in each diff corresponds to a parent of  $H$  in the dataflow graph.

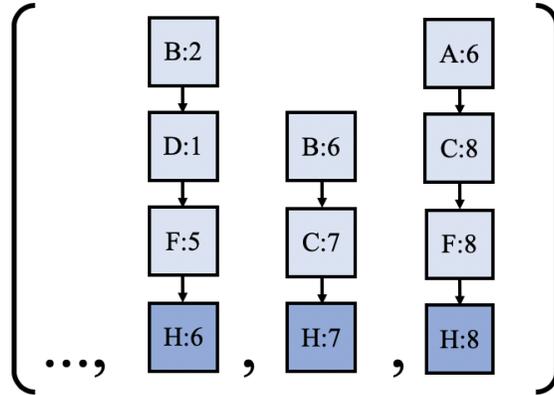


Figure 3-4: The diff log of node  $H$ , which contains all diffs that  $H$  has produced. Each diff has root  $H$ , and  $t_H$  increases sequentially across the log. Light blue represents the lineage that came from a parent, while dark blue indicates the lineage that came from  $H$ . The parent nodes in the diffs,  $F$  and  $C$ , are parents of  $H$  in the dataflow graph.

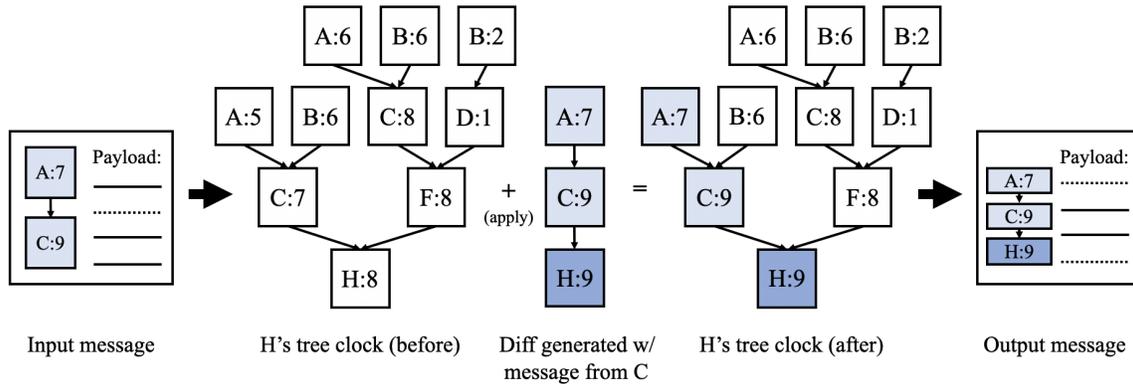


Figure 3-5:  $H$  receives a message from  $C$  with time  $t_C = 9$ .  $H$  adds a child node with time  $t_H + 1 = 9$  to the message diff, then applies the new diff to its own tree clock.  $H$  then sends a message with the transformed payload, including the last diff in the diff log.

### 3.1.4 Message processing algorithm

We now present the algorithm for how a node processes an input to produce an output, using the fields in Fig. 3-1. Initially, all times in the tree clock are 0 and all logs are empty. When  $H$  receives a message:

1. Copy the diff in the message, whose root is a parent node, and add  $H$  to the bottom of the diff with time  $t_H + 1$ , where  $t_H$  is from  $H$ 's own tree clock. Store the new diff in the diff log.
2. Apply the diff to  $H$ 's tree clock, which is an operation that takes the greater value in corresponding entries (Fig. 3-5).
3. Process the message payload, and include a copy of the diff in the output.

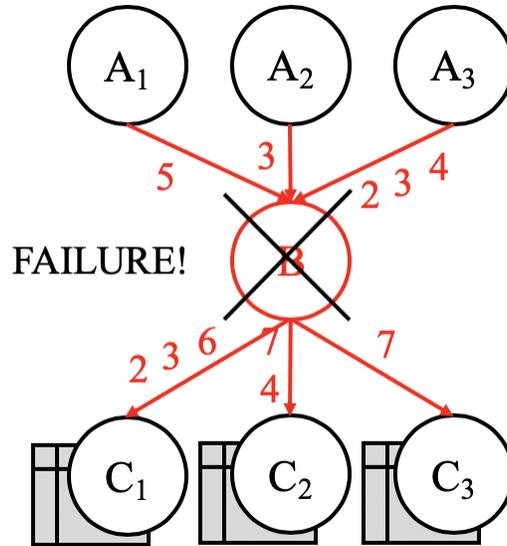


Figure 3-6: A failed node  $B$  with multiple parents and multiple children.  $B$  has  $m = 3$  parents  $A_1, A_2, A_3$  and  $n = 3$  children  $C_1, C_2, C_3$ . Even if  $B$  is a stateless node, we may lose messages that were in transit to and from the node, creating non-determinism when replaying messages on recovery.  $B$ 's children all have materialized state that would need to be purged if using Noria's original recovery protocol.

4. Send the output message and store the message in the payload log.

With this algorithm, if  $H$  fails, the surviving nodes have tracked enough information to be able to restore the system to a globally-consistent state. In particular,  $H$ 's children know which messages they received from  $H$ , where the messages came from, and in what order.  $H$ 's parents have the payloads to replay messages right where  $H$  left off. Now all that is required is a recovery protocol to put this information together.

## 3.2 Failure

In this section, we describe how the system utilizes the state on surviving nodes to restore the failed node to a globally-consistent state. A system-wide controller is responsible for officiating the entire recovery process. (§3.2.1). After detecting the failure, the controller aggregates the recovery-related metadata from surviving nodes to determine where each node should resume sending messages (§3.2.2). Finally, the restarted node must ensure its children receive each message exactly-once (§3.2.3).

### 3.2.1 Controller

The controller is responsible for detecting failures and officiating the recovery protocol. The controller detects the failure after a heartbeat timeout with a machine (Fig. 3-6). The controller determines which node was on the failed machine and restarts it on a working machine.

At this point, the controller needs to reform the network connections to and from the restarted node  $B'$  to integrate it back into the dataflow graph. Before it does, the controller tells the node's parents  $A_i$  not to send messages to  $B'$  until recovery is finished. This prevents  $B'$  from processing messages before its state has been initialized. Once the network connections are reformed, the controller begins the recovery protocol to determine where each node should resume sending messages.

It is important to distinguish the use of a system-wide controller, which is only invoked during failure, from global coordination in the normal case. Even during failure, the interactions between the controller and the dataflow graph are localized to a region including the failed node and its immediate neighbors, the *neighborhood*. We discuss neighborhoods more in §3.4.1. In addition, it is easier to reason about a complex recovery protocol that is officiated by a controller, as opposed to completely decentralized.

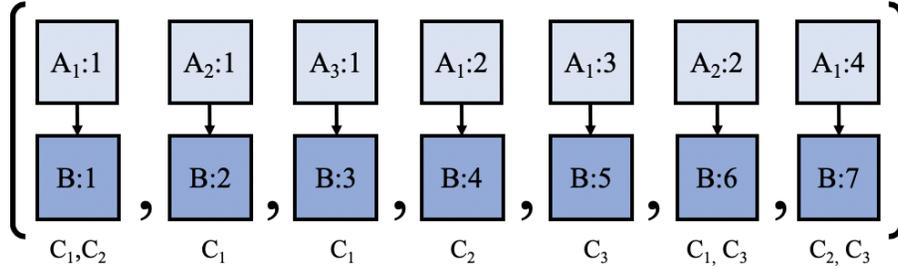
### 3.2.2 Recovery protocol

Once the controller has restarted the failed node on an existing machine and prepared the node's immediate neighbors to begin recovery, it begins the recovery protocol. The recovery protocol involves a series of message exchanges between the controller, the restarted node, and its immediate neighbors (Fig. 3-7):

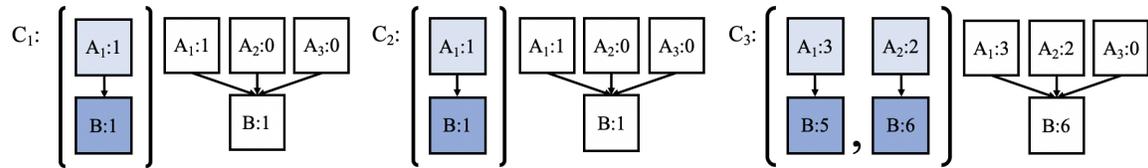
1. Ask all  $C_i$  for their diff logs and tree clocks rooted at  $B$  (Fig. 3-7b). Let  $t_{B,min}$  be the minimum  $t_B$  in the tree clocks. Calculate  $T^*$ , a tree clock with root time  $t_{B,min}$ , by applying all diffs up to and including  $t_{B,min}$  to a tree clock rooted at  $B'$  initialized with all zeros.
2. Tell  $B'$  to resume sending messages to each  $C_i$  at  $1 + B$ 's time in the clock from  $C_i$ , respectively (Fig. 3-7c). Include  $T^*$  for  $B'$  to initialize its tree clock. Wait for an ack.
3. Tell each  $A_i$  to resume sending messages to  $B'$  at  $1 + A_i$ 's time in  $T^*$  (Fig. 3-7c).

During the recovery protocol, the controller rolls back the recovery-related state in  $B'$  to a previously-consistent state of  $B$  without having to rollback downstream nodes. Note that  $T^*$  is not necessarily a state that  $B$ 's tree clock was actually in, since we only require  $B'$ 's recovered state to encapsulate the causal effects of the messages received by its children. The time  $t_{B,min}$  represents the latest time that  $C_i$  is guaranteed to have received a message from  $B'$ , if it should have received the message already. Also, the ack in step 2 is necessary to ensure that  $B'$  has already initialized its state with  $T^*$  before it receives any messages from  $A$ .

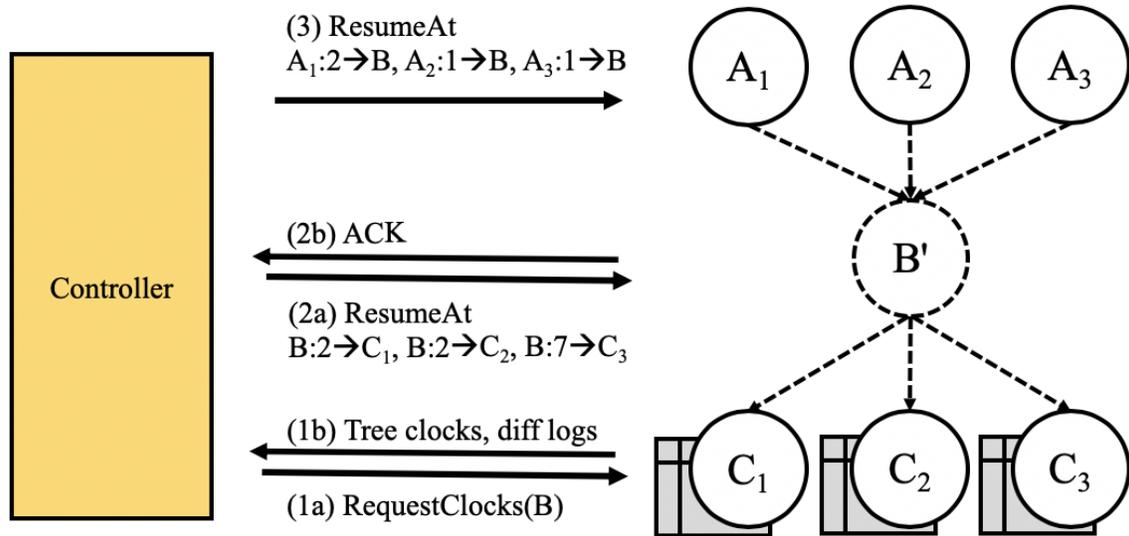
The controller might ask  $B'$  to resume sending messages that it had already sent before failure. However,  $B'$  just started up and has an empty payload log. As a result, it is unable to immediately resume sending messages to its children until it receives messages from its parents. If  $B'$  only has one parent, the order it receives messages is deterministic, and its computation is also deterministic. Analogously, if  $B'$  only has one child, the order it processes messages in does not matter since its child has not observed an ordering yet. However, if  $B'$  has multiple parents and multiple children, its children are able to observe a different interleaving of messages processed from the node's parents. Thus we require an



(a)  $B$ 's diff log from before the failure. Some of these messages are still in transit when the failure occurs (Fig. 3-6). The nodes underneath each diff indicate which children the message with that diff was sent to.



(b) In Step 1, each  $C_i$  sends its diffs and tree clock with root  $B$  to the controller.  $t_{B,min} = 1$  is the minimum  $t_B$  across all clocks.  $T^*$  is the result of applying diffs where  $t_B \leq t_{B,min}$  to a tree clock with all zeros. The only such  $t_B$  is 1, so  $T^* = B:1 [A_1:1 A_2:0 A_3:0]$ .



(c) In Step 2,  $B'$  resumes sending to  $C_1$  at 2,  $C_2$  at 2, and  $C_3$  at 7.  $B'$  sets its tree clock to  $T^*$ . In Step 3,  $A_1$  uses  $T^*$  to resume sending to  $B'$  at 2.  $A_2$  at 1.  $A_3$  at 1.

Figure 3-7: The recovery protocol officiated by the controller once it has restarted the failed node,  $B$ , on an existing machine as  $B'$ .  $A$  and  $C$  remain intact after the failure, including the materialized state in  $C$ . The diagram continues from the failure in Fig. 3-6.

### Invalid Execution from Processing Messages in the Received Order

Input	Output	Children	Valid?
$A_3:1$	$B':2$	$C_1$	Yes.
$A_1:2$	$B':3$	$C_2$	Yes.
$A_2:1$	$B':4$	$C_1$	Yes.
$A_1:3$	$B':5$	$C_3$	Yes, not sent to $C_3$ based on deduplication mechanism.
$A_3:2$	$B':6$	$C_2$	No, inconsistent with $B:6$ [ $A_2:2$ ] already received by its sibling $C_3$ .
$A_2:2$	$B':7$	$C_1, C_3$	No, $C_3$ already received the same message with a different diff $B:6$ [ $A_2:2$ ].

Figure 3-8: An invalid execution as the result of processing messages in the order they were received. The input column in the table reflects the order. The diagram continues from the failure in Fig. 3-6 and the recovery state machine in Fig. 3-7.

additional algorithm to ensure that the messages  $B'$  sends are consistent with those already received by its children (§3.3).

### 3.2.3 Deduplication

The restarted node needs to ensure that it does not send a message to a child that already received the message before the failure. To deduplicate messages,  $B'$  keeps a map from each child node ID to the minimum time it is okay to send to that child. By default, the minimum time is 0, but on recovery and until the next failure,  $B'$  sets the minimum time for each node to where it was told to resume sending messages to that node. Once  $B'$  has generated a message with a certain time, it checks for duplicates based on this map. Thus each  $C_i$  receives messages from the new  $B'$  right where the old  $B$  left off.

Alternatively, the receiving node  $C_i$  can deduplicate messages by discarding a message from  $B$  if  $t_B$  is not strictly greater than the root time of the previous message from  $B$ . This is more similar to the deduplication mechanism of other dataflow systems [2, 29]. However, we chose to deduplicate on the sending side because it requires one more read per message sent, rather than one more write per message received.

## 3.3 Ensuring a valid execution order

In addition to deciding where to resume sending messages, the system also needs to decide what order to process messages in to ensure the same interleaving of received messages as before the failure (Fig. 3-8). The order is important to ensure that downstream nodes, who may have states that depend on the input order, are consistent with each other. In this case, the restarted node rather than the controller decides the order based on an *execution replay* (*ER*) algorithm, since the processing order depends on the which messages were actually received. As discussed in §3.2.2, this section only applies to nodes with multiple parents

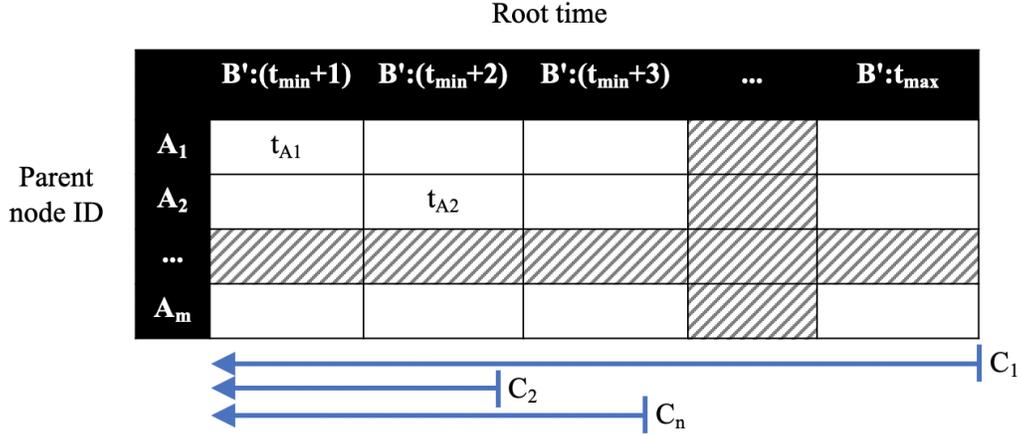


Figure 3-9: Partially completed table used to determine a valid execution replay. The first message that  $B'$  sends is  $B':(t_{min} + 1) [A_1:t_{A1}]$ , followed by  $B':(t_{min} + 2) [A_2:t_{A2}]$ . The  $C_i$  markers at the bottom of the table indicate that all messages left of that marker should have already been received by  $C_i$  or will never be sent to the node at all.

and multiple children.

The high-level goal of the execution replay is to resend all messages between  $t_{B,min}$  and  $t_{B,max}$ , since these messages have already been observed by some of  $B'$ 's children, but have yet to reach some of its other children. In order to resend these messages, the execution replay algorithm needs to figure out which upstream  $A_i$  each message came from, and how to send messages where this information is undetermined.

The post-recovery execution must reflect the ordering of diffs encapsulated in the diff logs of the restarted node's children. Each diff log describes a subset of messages that came from the failed node. The controller collects this information from each child and sends part of it to the restarted node. Let  $t_{B,max}$  be the maximum  $t_B$  in the tree clocks received from  $C_i$  in Step 1 of the recovery protocol (§3.2.2). The controller sends the diffs with root  $B$  where  $t_{B,min} < t_B \leq t_{B,max}$ . It is up to  $B'$  to determine what order to send them in once it has received the diffs. We call these *target* diffs because the restarted node must try to replay these exact diffs on recovery, while potentially sending different messages in between.

### 3.3.1 Execution replay table

We aid the process of determining an order to process received messages with the *execution replay table* (Fig. 3-9). A completed table defines what order  $B'$  should process messages in until it can return to normal operation.

We describe each aspect of the ER table and what it means in the context of replaying messages in order. The ER table belongs to the restarted node  $B'$ . Each column corresponds to a message with time  $t_{B'}$  that  $B'$  must replay, while each row corresponds to a parent node that  $B'$  might have received a message from. The integer entry in each column combined with the parent for that row correspond to the input message used to produce the output

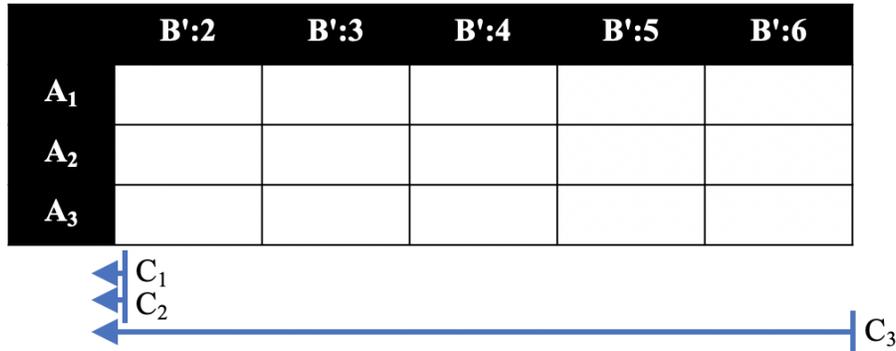


Figure 3-10: Empty ER table for the restarted node in Fig. 3-6. There is a column for each undetermined output message from  $t_{min} + 1 = 2$  to  $t_{max} = 6$ . There is a row for each parent  $A_1$ ,  $A_2$ , and  $A_3$ . There is a marker for each child  $C_1$ ,  $C_2$ , and  $C_3$  on where  $B'$  was told to resume sending messages in Fig. 3-7c.

message with time  $t_{B'}$ . Together, an entry  $t_A$  in row  $A$  column  $t_B$  means that  $B$  will send a message with diff  $B:t_B [A:t_A]$ . In addition, the markers at the bottom of the table indicate where  $B$  was told to resume sending messages to each child. The child  $C_i$  will only receive messages replayed after the time of the marker. To represent a *valid* execution, the solved ER table must fulfill several constraints:

- **Monotonicity constraint:** The entries in each row are in strictly increasing order.
- **Regularity constraint:** Each column has at most one entry, and columns that correspond to a target must have the matching entry.
- **Eventuality constraint:** If a message would be sent to  $C_i$ , the entry goes after the marker for  $C_i$ . If an entry is filtered, it may be discarded.

These constraints correspond to our correctness goals in §1.1. The Monotonicity and Regularity constraints ensure that updates are processed exactly-once and in the order they are sent by each parent. The Eventuality constraint ensures that each update is eventually processed, and thus the views are eventually-consistent.

The restarted node uses the diffs received from the controller to initialize the ER table (Fig. 3-10). For  $B'$  each row is one of its parents  $A_1$ ,  $A_2$ , and  $A_3$ . Each column is a message that  $B$  needs to send from starting from  $t_{B,min} + 1$ , one more than the root time of its tree clock, to  $t_{B,max}$ , the root time of the last target diff. Finally,  $B'$  uses the information from where it was told to resume sending messages to each child to create the markers at the bottom of the table.

### 3.3.2 Solving the ER table

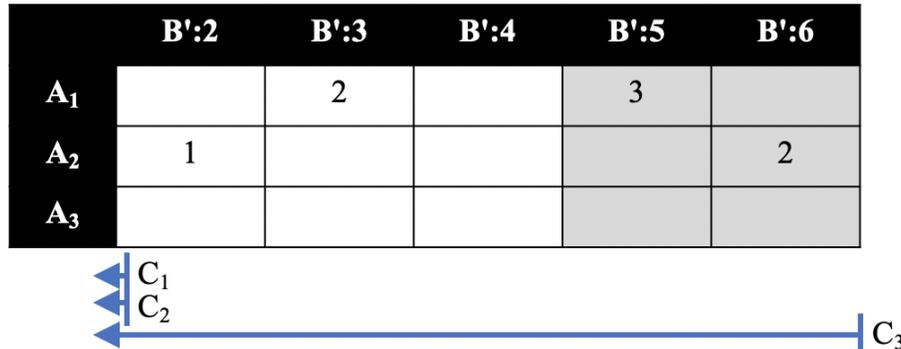
We must first collect the set of messages we can use to solve the ER table. Since the ultimate goal is to replay the target diffs,  $B'$  collects incoming messages in a buffer until it

### Algebraic Approach to Solving the ER Table

Input	Children	Monotonicity	Regularity	Eventuality	Result
$A_3:1$	$C_1$	$t \geq 2$	$t \neq 5, 6$	$t \geq 2$	No upper bound.
$A_1:2$	$C_2$	$2 \leq t \leq 5$	$t \neq 5, 6$	$t \geq 2$	$t \in \{2, 3, 4\}$
$A_2:1$	$C_1$	$2 \leq t \leq 6$	$t \neq 5, 6$	$t \geq 2$	$t \in \{2, 3, 4\}$
$A_1:3$	$C_3$		$t = 5$		$t = 5$
$A_3:2$	$C_2$	$t > t^{A_3:1}$	$t \neq 5, 6$	$t \geq 2$	No upper bound.
$A_2:2$	$C_1, C_3$		$t = 6$		$t = 6$

(a) Algebraic inequalities based on the Eventuality, Monotonicity, and Regularity constraints.

□ = Target



(b) Complete ER table.

Figure 3-11: Algebraic constraints on where each message can go in the table based on the received messages in Fig. 3-8 (Fig. 3-11). This diagram continues from the failure in Fig. 3-6. One valid solution to the table is to send  $B':2 [A_2:1]$  then  $B':3 [A_1:2]$  (Fig. 3-11b). However, any of the six solutions where  $A_2:1$  and  $A_1:2$  are placed in two different columns of  $B':2$ ,  $B':3$ , and  $B':4$  are valid. The targets are the diffs where  $t_{B,min} < t_B \leq t_{B,max}$ , in this case the diffs for  $B:5$  and  $B:6$ , which were observed by  $C_3$  but not its siblings. Messages that correspond to a target can only be placed in one position in the table.

has a message that corresponds to each target. A message corresponds to a target if its diff is the same as the target.  $B'$  then uses the buffered messages to fill out the ER table.

Any solution that satisfies the three constraints is sufficient, and there may be multiple possible solutions. There are also many different ways to find a solution to the constraint problem. At a high level, our method involves bounding the set of possible columns for each message based on the constraints, then narrowing down the space of solutions with a set of heuristics:

1. Place each message that corresponds to a target in the table.
2. Algebraically constrain where each message can go in the table based on the three table constraints (Fig. 3-11a). If there is no maximum bound, hold the message for later.
3. Place the remaining constrained messages in the table by brute-forcing every solution that satisfies the constraints (Fig. 3-11b).

In practice, we hope for the space of possible solutions to be relatively large, and for at least one solution to be easy to find. There must be at least one possible solution to the table, which is the execution of messages from before the failure. We can also optimize the brute-force checker using heuristics. For example, we can place messages with fewer possible solutions before placing others, or place messages in the order they were received as far left as possible. In a sense, solving the ER table is like a constrained topological sort.

### 3.3.3 Using the ER table to replay messages

The restarted node processes the messages in order from left to right in the completed ER table. Note that currently,  $B'$ 's tree clock is equal to  $T^*$ , which has root time  $t_{B,min}$  (§3.2.2). Thus based on the message processing algorithm in §3.1.4, the next message that  $B'$  sends will have time  $t_{B'} = t_{B,min} + 1$ . This is also the first entry in the execution replay table.

If the column has an entry, sending the message is straightforward. Let  $t_{A_i}$  be the entry in row  $A_i$  column  $B':t_{B'}$ . Then  $B'$  removes the input  $A_i:t_{A_i}$  from its buffered messages, processes it, and sends the output with diff  $B':t_{B'} [A_i:t_{A_i}]$ .

If the column is empty however, either there is a message we have no information about, or the message does not have an upper bound in the table. Regardless,  $B'$  pretends this space corresponds to a filtered message, generates a dummy message with that time to place in the payload log, and does not forward the message to anyone.

Once  $B'$  has reached  $t_{B,max}$ , the last message that a child had seen before the failure, recovery is complete and  $B'$  can send the remainder of its buffered messages as if they had just arrived. Note that the order that  $B'$  replays messages can easily be different  $B'$ 's original order (Fig. 3-12). However, the new order encapsulates the causal effects of previous messages that surviving nodes have observed.

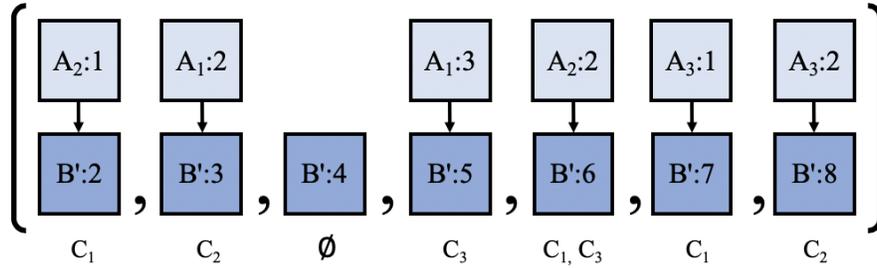


Figure 3-12:  $B'$ 's diff log after recovery based on the completed execution replay table in Fig. 3-11b. The order differs from  $B$ 's pre-failure order (Fig. 3-7a), but it is still valid because it is consistent with what its children have seen. In particular, it replays the messages with  $t_B = 5, 6$ , and otherwise meets the constraints specified by the execution replay table. Note that  $B'$  sends a dummy message for  $t_{B'} = 4$  because there is no entry for the corresponding column in the table.

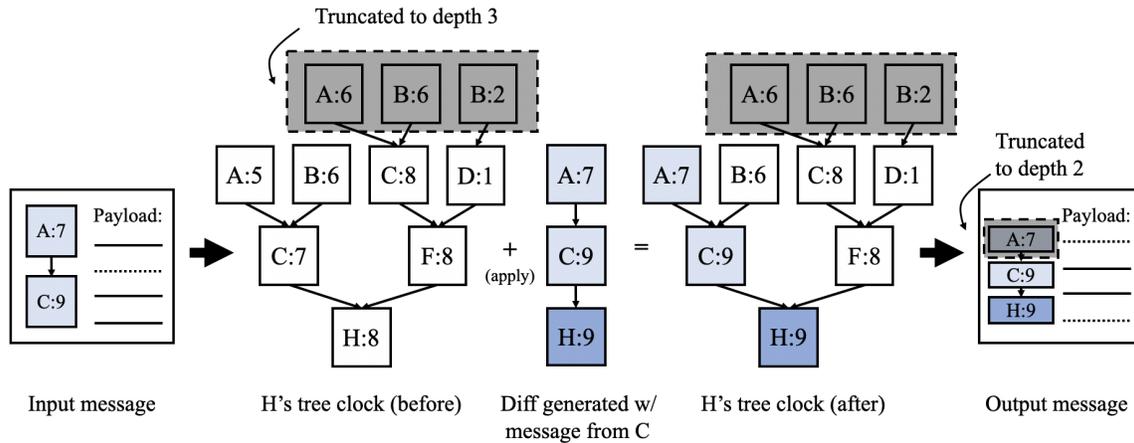


Figure 3-13: The apply diff operation from Fig. 3-5 with tree clocks of bounded size. The node's tree clock is bounded to depth three, while diffs are bounded to depth two before they are send with an output message.

## 3.4 Optimizations

### 3.4.1 Neighborhoods

As described, the sizes of tree clocks sent and stored in the system are bounded only by the size of the dataflow graph. Instead, we can bound them by the size of a *neighborhood*, a subgraph that includes a node and its immediate neighbors.

Note that in the recovery algorithm, the controller only communicates with the failed node and its neighborhood. In addition, the only nodes used in the state received from  $C_i$  are the nodes in the neighborhood. Thus in the tree clock algorithm in §3.1.1, we truncate each node's tree clock to depth 3 (Fig. 3-13). We also truncate diffs to depth 2 before sending them with a message.

A key insight to the practicality of tree clocks is that the diff is a constant-size data

structure. Even as the the graph scales out due to sharding, the diff remains linear, truncated to the depth of a neighborhood. This insight is particularly important because even though diffs are synchronously processed with every message in the system, we are able to keep the additional work small.

### 3.4.2 Log truncation

Unbounded logs are impractical without infinite storage. To decide where to truncate each log, let  $f(N)$  be the minimum time of ID  $N$  in the tree clocks across all nodes. A minimum bound for this value is calculated incrementally in the regular heartbeats that already exist for determining worker liveness.

Truncate the payload log of node  $N$  up to and including  $f(N)$ . Nodes downstream of  $N$  will have already received messages at least up to  $f(N)$ , so  $N$  will never be asked to send payloads less than  $f(N)$  in the event of a failure.

Truncate the diff log of node  $N$  with parents  $M_1, \dots, M_m$  up to but not including the first diff  $M_i:m$  where  $m \not\leq f(M_i)$ . Siblings of  $N$  will have already received messages at least up to the truncated diff, so  $N$  will never be asked to synchronize the truncated diffs. In addition, we must do log compaction on the diff log by applying truncated diffs to a base tree clock called the *min clock*. If we call the node's existing tree clock a *max clock*, then the two clocks and the diff log satisfy the following property:

$$\text{min clock} + \text{diffs} = \text{max clock}.$$

Diff log truncation affects Step 1 in §3.2.2, which assumes each node keeps a single tree clock. Instead, in Step 1, request both the min and max clocks from all  $C_i$ . Also, initialize  $T^*$  by applying all min clocks to each other, then apply diffs up to and including  $t_{min}$ . Another observation is that only children of multi-child multi-parent nodes need to keep a diff log at all, since as §3.2.2 describes, only failures of these types of nodes require an execution replay.

Sometimes, a node may have a very stale view of an ancestor  $N$ , preventing  $f(N)$  from being updated. For example, this may happen if  $N$  sends along an outgoing edge very infrequently, or if a data dependency prevents  $N$  from sending along specific edges at all. Therefore, we need some mechanism that recognizes when  $N$  has not updated its values along an edge within a certain time threshold, and periodically causes  $N$  to do so. This thesis does not provide a specific mechanism, but one possible solution would be for  $N$  to track this information itself.

# Chapter 4

## Implementation

We implement a prototype of tree clocks and the tree clock recovery protocol on Noria written in 4k lines of Rust.

*TreeClock* is a recursive data structure of node IDs and times, with edges to other *TreeClocks* (Fig. 4-3). In the implementation, a node in a tree clock is a single-threaded connected subgraph of dataflow nodes. The abstraction still holds because messages are serialized at the inputs and outputs, and computation is deterministic. A sharded subgraph is considered to be many distinct nodes, each with its own ID. *Diff*s are a subset of linear tree clocks, so we alias the type with the *TreeClock* data structure.

Each node keeps additional fields for recovery-related bookkeeping (Fig. 4-2). *min\_time* and *payloads* are used as a payload log with log truncation. *store\_diffs*, *min\_clock*, *max\_clock*, and *diffs* are used as a diff log with log compaction and an option to not store diffs at all. *do\_not\_send* prevents  $A_i$  nodes from sending to  $B'$  before recovery is complete, while *min\_time\_to\_send* prevents  $B'$  from sending duplicates to its children. *targets* and *buffer* are used in the execution replay.

The recovery protocol is implemented as a series of message exchanges between the controller, the failed node, and its immediate neighbors. The *RequestClocks* message corresponds to Step 1 in the recovery protocol (§3.2), where the controller asks  $C_i$  for their diffs and tree clock rooted at a node ID. The *ResumeAt* message corresponds to Steps 2 and 3, where the controller may also include  $T^*$  and a list of target diffs for  $B'$  to use in recovery.

---

```
1  struct TreeClock {
2      root: NodeID,
3      time: usize,
4      edges: HashMap<NodeID, Box<TreeClock>>,
5  }
6
7  type Diff = TreeClock;
```

---

Figure 4-1: The clock data structures in the implementation. The *Diff* type is aliased as a *TreeClock* because it is a subset of linear tree clocks.

Field	Rust Type	Section	Purpose
payloads	Vec<Box<Packet >>	§3.1.2	Payload log.
min_time	usize	§3.4.2	Payload log truncation.
diffs	Vec<Diff>	§3.1.3	Diff log.
max_clock	TreeClock	§3.1.1	Primary tree clock.
min_clock	TreeClock	§3.4.2	Diff log compaction.
store_diffs	bool	§3.2.2	Option to not store diffs at all.
do_not_send	HashSet<NodeID>	§3.2.1	Prevent parents from sending to restarted node until recovery is complete.
min_time_to_send	HashMap<NodeID, usize>	§3.2.3	Deduplication.
targets	Vec<Diff>	§3.3	Target diffs.
buffer	HashMap<NodeID, Vec<(usize, Box<Packet>)>>	§3.3.2	Buffer messages for solving the ER table.

Figure 4-2: Recovery-related bookkeeping for each node in the implementation.

```

1  enum Packet {
2      // Request the diffs and tree clocked rooted at the
3      // given node ID, which is the ID of a parent node
4      RequestClocks (NodeID) ,
5
6      // Notify the domain where to resume sending
7      // messages to its children
8      ResumeAt {
9          id_times: Vec<(NodeID, usize)>,
10         min_clock: Option<TreeClock>,
11         targets: Vec<Diff>,
12     },
13 }

```

Figure 4-3: The messages exchanged in the recovery protocol (§3.2.2), in the implementation. *RequestClocks* is used in Step 1 of the recovery protocol, while *ResumeAt* is used in Steps 2 and 3.

# Chapter 5

## Evaluation

The message processing algorithm requires additional computation at each node crossing to update clock times. There is also computation involved in managing and truncating the logs. We want to analyze how the algorithm affects performance in the normal case and recovery time in the failure case by answering the following questions:

1. How do tree clocks affect the staleness of reads? (§5.2)
2. How do tree clocks affect read and write latency? (§5.3)
3. How does the recovery time compare to original Noria? (§5.4)

### 5.1 Benchmarking methodology

#### 5.1.1 Dataflow graphs

We use a query based on the Lobsters news aggregator from the original Noria paper [16] in our performance benchmarks (Fig. 5-1). The resulting dataflow graph consists of a stateless sharder node with multiple stateful parents sharded by article ID and multiple stateful children sharded by author ID (Fig. 5-2). Given an input message, the sharder node can send a message to any number of its children in the graph.

During setup, the load generator prepopulates the *Article* base table with a variable number of articles, uniformly assigned to one of 400 authors. During the benchmark, the load generator queues writes to *Vote* and reads to *AuthorWithVoteCount* according to a target load. Writes are keyed by a random article ID while reads are keyed by a random author ID.

We measure write propagation time using a reserved article ID and author ID key. Writes to the reserved key are only performed when we want to get a data point for write propagation time, and this key is not included in the random article ID generator. Immediately after sending the write, we perform reads for the corresponding author ID of the reserved key until the vote count has gone up, indicating the write has propagated. The granularity with which we can measure propagation time is the same as the batch interval with which we execute queued requests.

---

```

1 # base tables
2 CREATE TABLE Article
3     (id int, title varchar(255), author_id int, PRIMARY KEY(id));
4 CREATE TABLE Vote (article_id int, user int);
5
6 # read queries
7 CREATE VIEW VoteCount AS
8     SELECT Vote.article_id, COUNT(user) AS votes
9     FROM Vote
10    GROUP BY Vote.article_id;
11 CREATE VIEW ArticleWithVC AS
12     SELECT Article.id, author_id, VoteCount.votes AS votes
13     FROM Article
14    LEFT JOIN VoteCount
15    ON (Article.id = VoteCount.article_id)
16    WHERE Article.id = ?;
17 QUERY AuthorWithVC: SELECT author_id, SUM(votes) as votes
18     FROM ArticleWithVC
19    WHERE author_id = ?
20    GROUP BY author_id;

```

---

Figure 5-1: The SQL tables and queries used in the benchmark. In a news aggregator, we have a table of articles, and a table of votes for the articles. The *VoteCount* query tallies the votes by article ID, and the *ArticleWithVC* query combines this tally with the author of the article. The *AuthorWithVC* query is interested in the number of votes an author has received across all the articles written by that author, which can be computed as a result of the *ArticleWithVC* query.

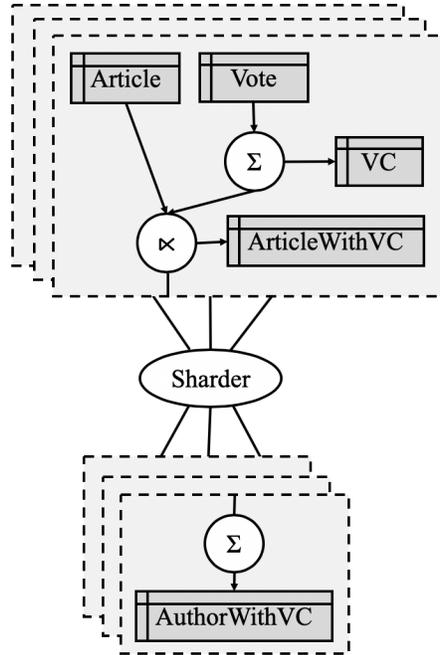


Figure 5-2: The sharded dataflow graph defined by the SQL in Fig. 5-1. The inputs to the sharder node are a graph sharded by article ID, and the outputs of the sharder node are a graph sharded by author ID. During the benchmark, we send writes to *Vote* and reads to *AuthorWithVC*.

We use the following default parameters, unless otherwise stated. We deploy a 20-way sharded graph, meaning the sharder node has 20 parents and 20 children, and prepopulate the base tables with 1 million articles. The load generator uses a write-heavy workload of 50% reads and 50% writes at 10000 ops/s. Writes are batched only at the input node with a batch interval of 1000 $\mu$ s. Each data point is collected over a 30 second trial.

### 5.1.2 Hardware

All experiments run on a machine with 40 CPUs. It is impractical to run a deployment with more than 20 shards since the performance gains of sharding rely on having enough cores to properly parallelize computation. Also, the machine has 62.8GB of memory, which places a limit on the amount of state we can accumulate in the table for the benchmark. Although we did not have the resources to do so for this thesis, we plan on exploring larger sharding factors and larger state sizes in future work by deploying Noria across multiple machines.

## 5.2 Read staleness

We quantify the runtime overhead of tree clocks by measuring the time it takes writes to propagate through the graph. Noria clients observe write propagation time as the staleness of a read. Since the bookkeeping of tree clocks is proportional to the number of node cross-

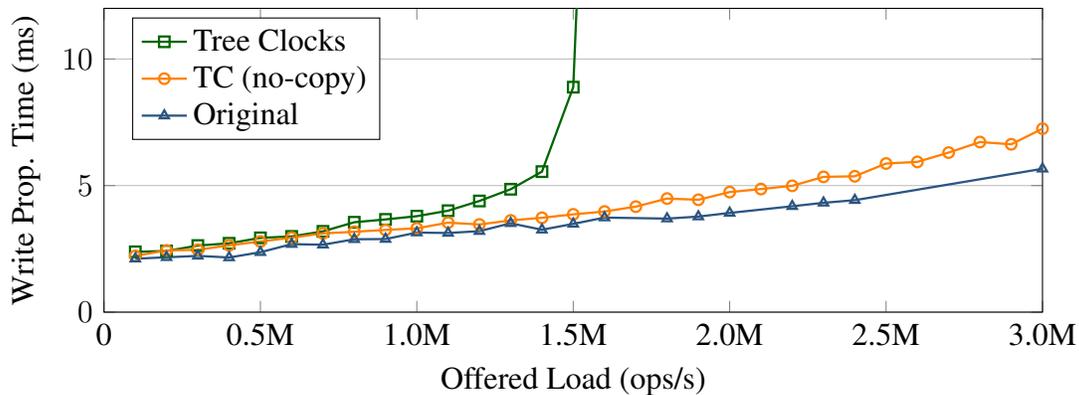


Figure 5-3: Offered load versus write propagation time. At 1.2 million ops/s with 4ms of write propagation time, the overhead of tree clocks is 37% with the copy and 8% without. At 3.0 million ops/s the overhead of tree clocks without copy is 28%. In absolute numbers, the difference in write propagation time at 3.0 million ops/s is 1.5ms.

ings, write propagation time measures the compounded overhead as writes flow completely through the graph.

A substantial amount of overhead comes not from the bookkeeping associated with tree clocks, but from copying the message into the payload log, which we address in the next paragraph. Near the maximum load of 1.2 million ops/s with 4ms of write propagation time, the overhead of tree clocks is 37% with the copy and 8% without (Fig. 5.2). At 3.0 million ops/s the overhead of tree clocks without copy is 28%. In absolute numbers, the difference in write propagation time at 3.0 million ops/s is 1.5ms.

**Tree clocks no-copy.** As discussed, a substantial amount of overhead comes not from the bookkeeping associated with tree clocks, but from copying the message into the payload log. Since messages are batched, as the offered load increases, the total number of messages in the system does not change, but the size of the message payload increases (Fig. 5-4).

However, this problem is not fundamental to tree clocks, as the diff size stays constant regardless of the offered load (Fig. 5-4). We will address the problem in future work by tracking a reference to the payload instead of copying it. For benchmarking purposes, we collect data on two versions of Noria with tree clocks: one that copies the payload into the log, and another that omits the payload but still manages logs.

In addition, offered loads beyond the saturation point of tree clocks with-copy do not reflect a practical Noria deployment. At the saturation point of 1.5 million ops/s, at least half the messages are sent to every shard because the payload is so large (Fig. 5-5). A load this high does not effectively exploit the parallelism of sharding, and in a practical deployment we would increase the number of shards and distribute them over multiple machines.

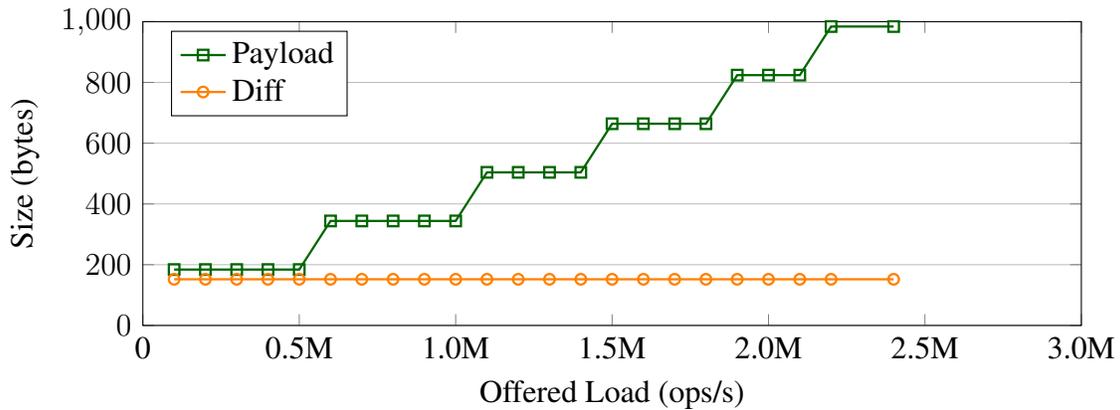


Figure 5-4: Offered load versus the median size of message components leaving the sharder node. The size of the payload increases in steps for each additional row of data in the median message, while the size of the diff remains constant. Since we currently copy the entire payload into the payload log, this results in significant overhead as the sizes of messages increase. However, we believe this is an implementation detail that can be solved by tracking a reference to the payload instead of copying it.

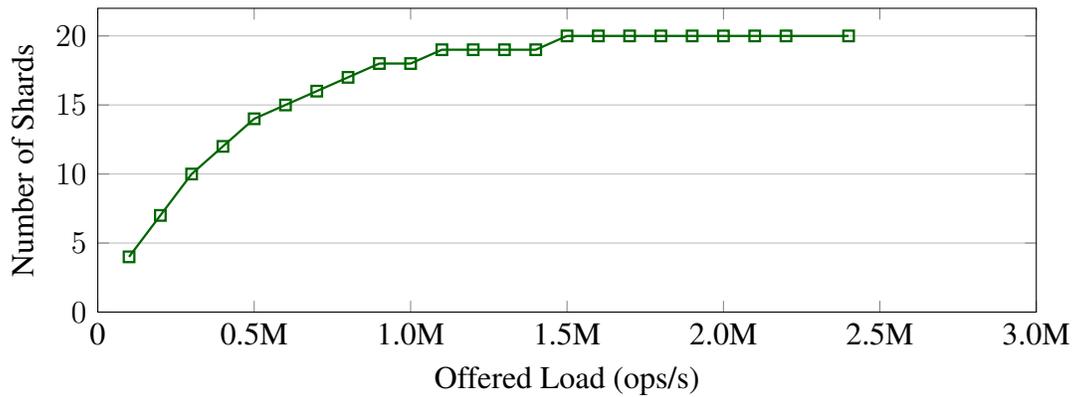


Figure 5-5: Offered load versus the median number of shards a sharder splits an outgoing message into in a 20-way sharded graph. These results indicate that the system is fully saturated at about 1.5M ops/s, where the sharder sends a message to every shard. At this point, in a practical deployment we would increase the number of shards to fully exploit the parallelism of multiple cores. We may also need to distribute the computation over more machines to attain more cores.

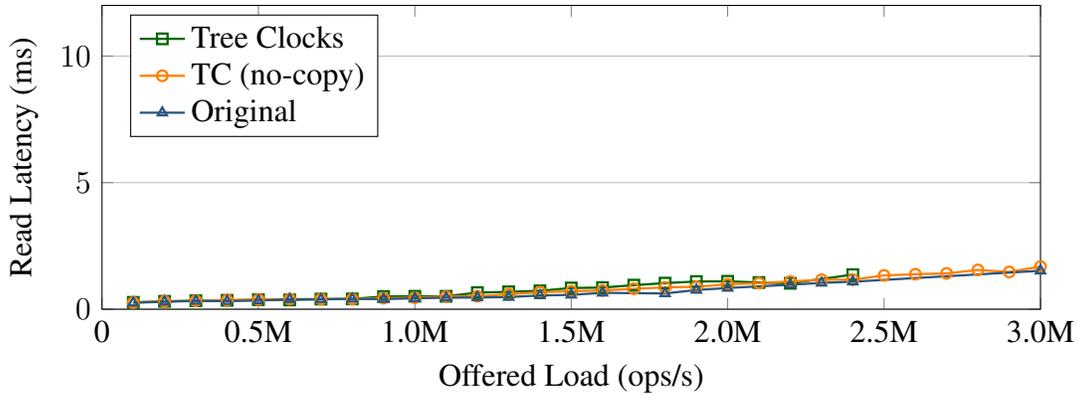


Figure 5-6: Offered load versus the time it takes for a read to return a cached value. There is no significant difference with or without tree clocks.

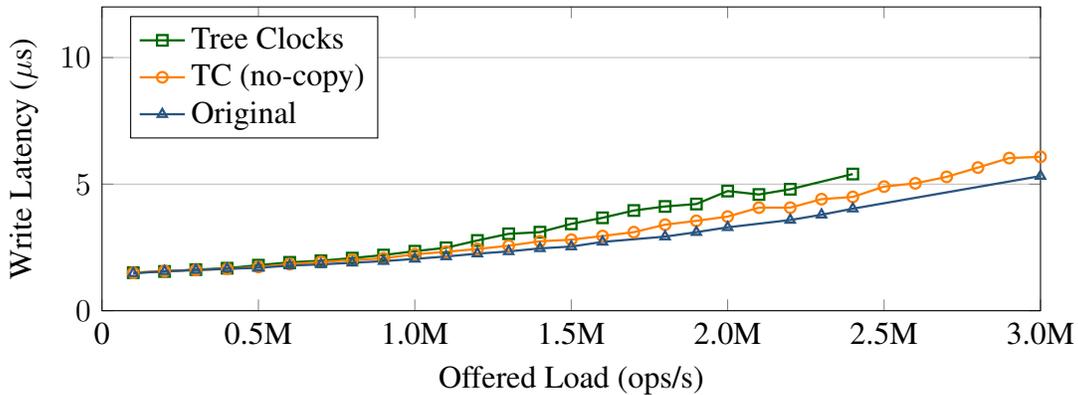
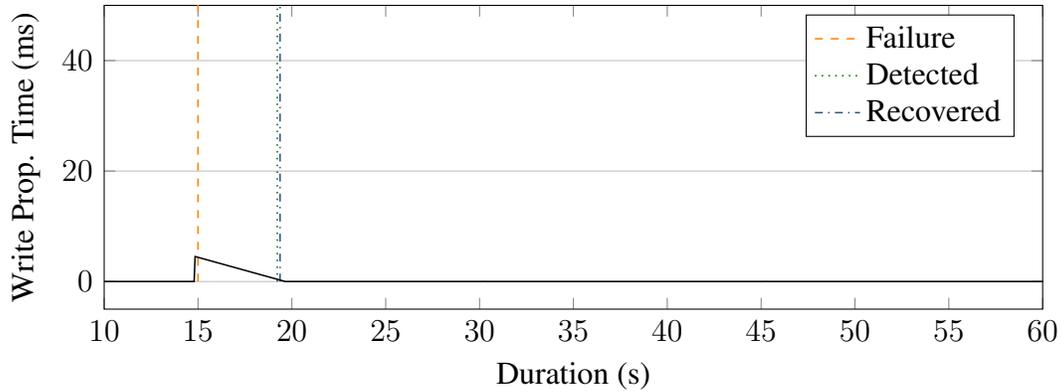


Figure 5-7: Offered load versus the time it takes to inject a write into the dataflow. The write latency of tree clocks is greater than that of tree clocks no-copy, which is greater than that of original Noria. At 2.4 million ops/s, the overhead of TC no-copy is about 0.5 milliseconds.

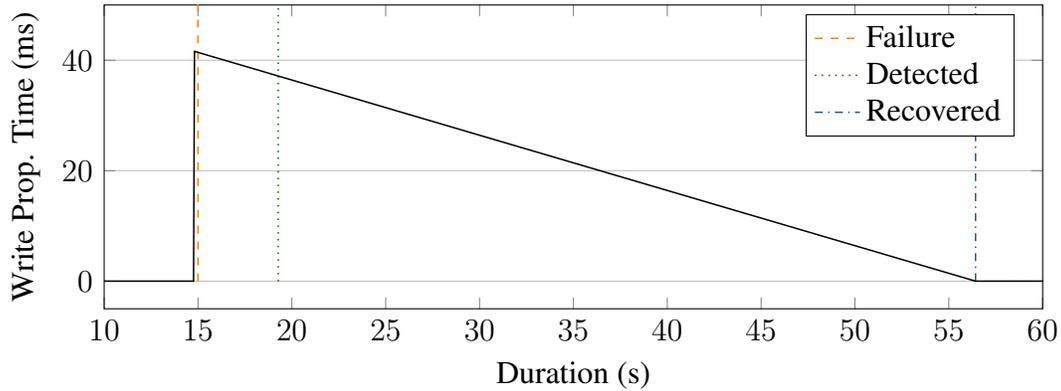
### 5.3 Read and write latency

We measure read latency as the time it takes for a read to return a value (Fig. 5-6). Read latency remains the same with tree clocks, since Noria can immediately return a cached value even if writes have not fully propagated through the graph.

We measure write latency as the time it takes the first node in the dataflow graph to process and send the write message, then return a response (Fig. 5-7). We define write latency this way because writes are acked they moment they become durable, which is when they enter the graph. Unlike write propagation time, write latency does not depend on the depth of the graph, and the value reflects the overhead of a single node crossing. At this level, message diffs are also smaller because the first node does not have parents. The overhead of tree clocks remains relatively low, but increases proportionally to the offered load for reasons we will investigate in future work. Near the maximum load of 2.4 million ops/s, the overhead of write latency is about 0.5 milliseconds.



(a) Neighborhood recovery algorithm; detected=19.223s, recovered=19.372s.



(b) Original recovery algorithm; detected=19.275s, recovered=56.425s.

Figure 5-8: Write propagation over time for each recovery protocol with 50 million articles. The failure occurs at 15s, and the controller detects the failure a few seconds later after a heartbeat timeout. The difference between recovery and detection time in the neighborhood recovery algorithm is almost negligible. In comparison, recovery in the original Noria is more than 30 seconds.

## 5.4 Recovery time

We measure recovery time by killing one of the nodes in the benchmark and measuring the time it takes for the system to return to normal operation. To do so, we probe the write propagation time every 250ms, where an abnormally large time indicates the graph is disconnected and writes are unable to propagate (Fig. 5-8). The recovery time does not include the time it takes to detect the failure, which is independent of the recovery protocol.

The system is considered to be in normal operation once the write propagation time reaches pre-failure levels again. This is a conservative approach in tree clocks, here one might also consider the system to be recovered as long as the graph is fully-connected, but still handling a backlog of writes from when the node was offline.

In the original recovery algorithm, reads become unavailable while the system is still

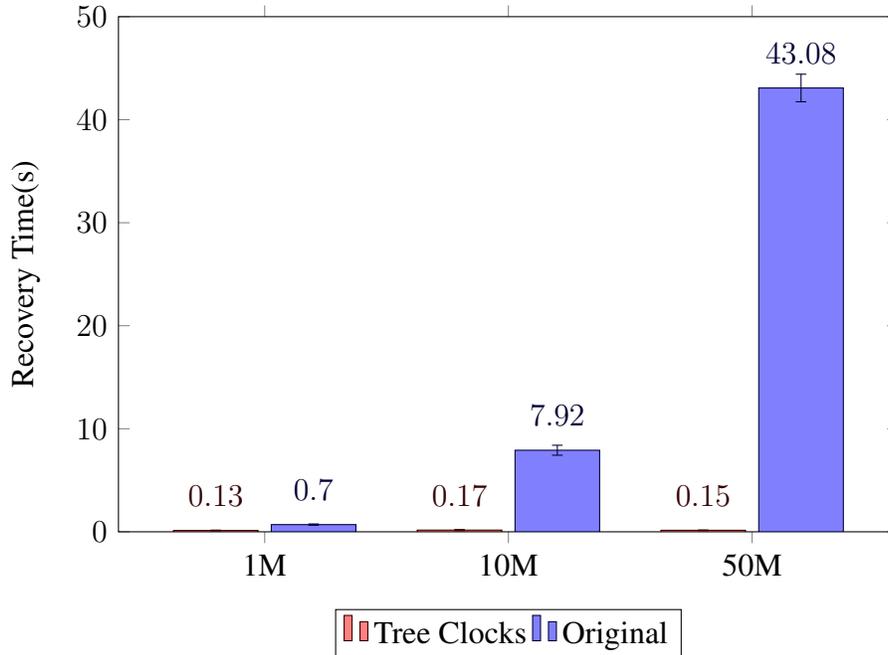


Figure 5-9: Number of articles versus recovery time as the median of 11 trials. The error bars show the recovery times at the first and third quartiles. The recovery time for tree clocks is constant in relation to the number of articles. In comparison, the recovery time in the original Noria is proportional to the number of articles. For example 10M articles at 7.92s is about 10x the recovery time for 1M articles at 0.7s. Similarly, 50M articles at 43.08s is about 60x.

recovering. This is because the original protocol drops and restarts the reader node, which has to rebuild its entire state (without partial materialization) until it can respond to read requests again. However, this as an implementation detail, since Noria could theoretically keep the old reader node around to return stale values until the new reader node is ready to respond to requests.

In the tree clock recovery algorithm, reads return stale values, but stay available throughout the failure. This is because the only node that tree clock recovery restarts is the failed node. The client handle to the view stays the same, and the reader can continue to respond with stale values until recovery is completed. It is important to note that both reads and writes remain online throughout the duration of the failure, a vital property for web applications where availability is key.

We compare the recovery times from losing the sharder node, which would require re-processing all rows in the `Article` base table if rebuilding state (Fig. 5-9). The recovery times are collected as the median over 11 trials. With 50 million articles, tree clock recovery took 0.15s, a 290x improvement from the original Noria, which took 43.08s. Note that 0.15s is below the 250ms granularity with which we measure write propagation time, but is sufficient to demonstrate the advantages of tree clocks. We did not collect data beyond 50 million articles since we approached the memory limits of our machine, but we expect real Noria deployments to use multiple machines to store their base tables.

The key takeaway about recovery time is not the large multiplicative factor, which does not carry much meaning in itself, but that we have eliminated the dependence of recovery time on the amount of state in the system. As systems accumulate more data over time, rebuilding state in the original Noria can only take longer than 43.08s. In comparison, the recovery time using tree clock recovery is independent of the number of articles in the system and remains constant.



# Chapter 6

## Correctness

Noria is eventually-consistent and exactly-once. We describe the state of the system after recovery to be correct if it is a state it could have reached had no failure happened at all. In particular, we must resolve the non-deterministic order that the recovered node receives messages to be consistent with the order observed by its surviving downstream nodes. The state we consider includes: the min and max clock, the payload log, and the diff log.

First, we introduce a model for describing a node's execution. We then constrain the model with a set of invariants to describe only valid executions. Next, we prove that the recovery algorithm re-creates an execution where these constraints still hold despite a failure. Finally, we prove that the recovery-related bookkeeping in the system will eventually be ready to handle the failure of another node.

### 6.1 Invariants

**Definition 1.** Given a series of inputs to a node  $[IN_1, IN_2, IN_3, \dots]$ , we can model the execution of the node as a deterministic state machine:

$$\begin{aligned} \text{state}_0 &= \text{EMPTY} \\ \text{state}_i &= f(\text{IN}_i, \text{state}_{i-1}) \\ g(\text{IN}_i, \text{state}_i) &= \text{OUT}_i \\ h(\text{OUT}_i) &= \text{children}_i, \end{aligned}$$

where  $f, g, h$  are deterministic functions. In stateless nodes,  $\text{state}_i = \text{state}_0$  for all  $i$ .

The crux of the recovery algorithm is determining which inputs  $B'$  and  $C_i$  should receive next, and in what order. In particular,  $B'$ 's inputs determine  $B'$ 's outputs, and subsequently  $C_i$ 's inputs. The execution would trivially be valid if post-recovery,  $B'$  and  $C_i$  received the exact same inputs as the ones they would have received pre-failure. However, we relax this solution by allowing *any* valid execution that reflects the causal effects of previous messages, which is an execution that satisfies:

- **Monotonicity Invariant:** Inputs from a parent A are received in strictly increasing, though not necessarily sequential, order.

- **Regularity Invariant:** If two nodes receive inputs with diffs  $A:t_A [DIFF_i]$  and  $A:t_A [DIFF_j]$ , respectively, then  $DIFF_i = DIFF_j$ .
- **Eventuality Invariant:** If  $B \in h(\text{OUT})$  for some output of  $A$ , then  $B$  should eventually receive and process that output as an input.

These invariants follow from the definition of the message processing algorithm and how nodes generate a diff to send with each message (§3.1.4). Additionally, messages are sent over reliable, ordered TCP streams.

## 6.2 Proof: the recovery protocol produces a valid execution order.

Consider the general case of a failed node  $B$  with  $m$  parents  $\{A_i, i \in [1, m]\}$  and  $n$  children  $\{C_i, i \in [1, n]\}$ , without log truncation. The recovery protocol restarts the failed node on a new computer as  $B'$ . We will prove that the recovery algorithm re-creates an execution where the invariants still hold despite a failure.

### 6.2.1 Each $C_i$ 's inputs reflect a valid execution after recovery.

We make the important assumption that  $B'$ 's post-recovery outputs reflect a valid series of inputs from its parents, which we prove in §6.2.3. In this proof, we focus exclusively on the relationship between  $C_i$  and its immediate neighbors.

**Lemma 2.** The invariants hold in  $C$ 's post-recovery execution when  $C$  is  $B$ 's only child and  $B$  is  $C$ 's only parent.

Assume  $C$  is  $B$ 's only child and  $B$  is  $C$ 's only parent. Let  $B:t_B$  be the last input  $C$  received from  $B$ . Then  $C$  has a diff in the diff log of the form  $C:t_C[B:t_B]$ . If  $C$  is supposed to receive any messages  $B':t_{B'}$  from the restarted node where  $t_{B'} \leq t_B$ , it would have already received them due to Monotonicity. Thus  $C$  needs to receive messages from  $B'$  where  $t_{B'} > t_B$ .

$B'$ 's max clock is the result of all of  $C$ 's diffs with root  $B$  applied to  $B'$ 's min clock (§3.2.2). (Without log truncation, the min clock is all zeros.) Thus  $t_{B'}$  in the max clock is the greatest time corresponding to  $B$  in  $C$ 's diff log  $\Rightarrow t_B$ .  $B'$  resumes sending messages to  $C$  at 1 more than  $t_{B'}$  in  $B'$ 's max clock  $\Rightarrow t_B + 1$ . Thus Monotonicity holds.

It follows that  $C$  will eventually receive any messages  $B':t_{B'}$  where  $t_{B'} > t_B$  since this is equivalent to  $t_{B'} \geq t_B + 1$ . Thus Eventuality holds.

$B'$  does not have other children, so no other nodes can receive an input from  $B'$  with the same time. Thus Regularity also holds.

**Corollary 3.** WLOG, the invariants hold in  $C_i$ 's post-recovery execution where  $B$  has any number of children  $\{C_i, i \in [1, m]\}$ , but  $B$  is  $C_i$ 's only parent.

Assume  $B'$  started generating messages at  $t_{B'} < t_B + 1$  instead of at  $t_B + 1$ , but  $B'$  knows that  $C_i$  should only receive outputs  $\text{OUT}_t$  where  $t \geq t_B + 1$ . Then when generating

$\text{OUT}_{t_{B'}}$ , even if  $C_i \in h(\text{OUT}_{t_{B'}})$ ,  $B'$  will not send the message to  $C_i$  due to  $B'$ 's deduplication mechanism (§3.2.3). Thus the time  $B'$  resumes sending message to each  $C_i$  is a maximum bound on when  $B'$  can start *generating* messages while satisfying Eventuality. The minimum maximum bound is  $t_{B,\min} + 1$ , as in the algorithm (§3.2.2).

Now, assuming WLOG  $C_i$  has multiple parents including  $B$ ,  $C_i$  receives messages in strictly increasing order from each parent. From the perspective of  $C_i$ 's children, the ordering of these messages has not been decided yet, thus any interleaving is valid. It follows from Lemma 2 and Corollary 3 that  $C_i$ 's post-recovery execution is valid.

## 6.2.2 $B'$ 's inputs reflect a valid execution after recovery.

During the execution replay,  $B'$  decides on an ordering of its inputs (§3.3). We define the inputs to reflect a valid execution if they contain all the messages required to be ordered as a valid execution.

**Lemma 4.**  $T^*$  is the value of  $B$ 's max clock in some valid execution. That is, an execution that reflects the causal effects of previous messages.

We obtain  $T^*$  by applying all diffs up to and including  $t_{B,\min}$  to a tree clock with all zeros (§3.2.2). If this includes all diffs from 1 to  $t_{B,\min}$ , then  $T^*$  is trivially a max clock. However, it is possible that we do not apply some diff  $B:t_B^*$ . This would occur if no child  $C_i$  received a message from  $B$  with time  $t_B^* < t_{B,\min}$ . However, each child has received some message from  $B$  with time  $t_B \geq t_{B,\min}$  or  $t_{B,\min}$  would not be the minimum across all max clocks. Thus the message with time  $t_B^*$  was not sent to any children, or it would have already been received due to Monotonicity.

The valid execution that this max clock reflects is one in which all diffs that have been received by some  $C_i$  are in  $B'$ 's diff log. Since  $B'$  is stateless, regardless of when  $B'$  sends the message  $t_{B'}^* = t_B^*$ , the message will deterministically not be sent to any children and  $C_i$ 's inputs are not affected. Thus in this execution, the filtered messages may or may not have been sent already.

Given  $T^*$  as  $B'$ 's max clock, we will prove that each  $A_i$  sends the messages required for  $B'$  to order them as a valid execution. Let  $t_{A_i}^*$  be the time in  $T^*$  for some parent  $A_i$ . Then  $B'$  has already “received“ all messages  $t_{A_i}$  from  $A_i$  where  $t_{A_i} \leq t_{A_i}^*$  due to Monotonicity.  $A_i$  resuming sending all messages  $t_{A_i} > t_{A_i}^*$  (§3.2.2), which is equivalent to resuming at  $t_{A_i}^* + 1$ . Additionally, we can assume that  $A_i$ 's post-recovery outputs are valid since they are either the pre-failure outputs stored in the payload log, or were just generated.

Each  $A_i$  sends all messages to  $B'$  that have not yet been applied to  $B'$ 's max clock, as long as  $B' \in h(\text{OUT})$ . This includes messages that  $B'$  received before the failure but filtered to its children. Since the execution replay algorithm (§3.3) allows discarding filtered messages,  $B'$  can theoretically recreate the same order of inputs as before the failure.

### 6.2.3 $B'$ 's outputs determine valid inputs to each $C_i$ after recovery.

The proof in §6.2.1 depended on this vital assumption. While §6.2.2 guarantees that we have the messages to potentially regenerate a valid execution, this section proves we have sufficient information to constrain the actual ordering.

**Lemma 5.** The execution replay constraints in §3.3 each guarantee an invariant in §6.1.

Regularity holds if  $B'$  can buffer a message corresponding to each target.  $B'$  must eventually receive the corresponding input because the inputs received from a parent are deterministic, in strictly increasing order, and will eventually arrive.

Eventuality holds if  $B'$  can place every received message, except filtered messages, in the ER table to be sent to its intended children. Let  $t_B$  be the root time of the last target provided by  $C_i$ . This time is equivalent to the marker for  $C_i$  in the ER table (Fig. 3-9). If there exists some output OUT placed at a time  $t \leq t_B$ , where  $C_i \in h(\text{OUT})$ , then  $B'$  would filter the message to  $C_i$  as a duplicate (§3.2.3). Thus  $t$  must be placed at a time  $t > t_B$ , a minimum bound for  $C_i$  to eventually receive the message.

Monotonicity holds if  $B'$  places an input  $A_i:t_{A_i}$  in row  $A_i$ , where  $t_{A_i}$  satisfies Monotonicity relative to the other entries in that row. This is a maximum bound if and only if there is an entry  $t$  in row  $A_i$  where  $t > t_{A_i}$ . The minimum bound may or may not be stronger than the Eventuality minimum bound.

**Lemma 6.** There exists a solution that satisfies these three constraints.

Let  $t_{B'}$  be the root time in  $B'$ 's max clock,  $T^*$ . From §6.2.2,  $B'$  receives all messages that  $B$  would have received after when  $B'$ 's root time was  $t_{B'}$ , and also some messages that  $B$  received before this time and filtered. But since our Eventuality constraint does not require filtered messages to be placed in the table, there must be at least one solution to the execution replay table, which is the solution that produces the pre-failure execution order.

Lemma 5 guarantees that as long as  $B'$  can find a solution to the execution replay table, then the solution corresponds to a valid execution order. Lemma 6 guarantees that at least one solution exists. This section omits the proof that our proposed method for solving the execution replay table (§3.3.2) finds a valid solution, which we believe is intuitive from the description of the method.

### 6.2.4 The recovery protocol is correct with log truncation.

We will prove that log truncation does not affect the correctness of the recovery protocol.

Assume FSOC the controller asks  $A$  to resume sending messages to  $B'$  at  $t_A^*$ , except  $A$  has already truncated payload  $t_A^*$ . According to the truncation algorithm,  $t_A^* \leq f(A)$ , where  $f(A)$  is at most the  $t_A$  in the tree clocks across all nodes (§3.4.2). Thus  $t_A^* \leq f(A) \leq t_A$  in each of  $C$ 's max clocks. Let  $t_{A,min}$  be the minimum of these values for  $t_A$ . It follows that  $t_A^* \leq t_{A,min}$ . The algorithm tells  $A$  to resume sending messages at  $t_{A,min} + 1 > t_A^*$  (§3.2.2), a contradiction. Thus  $A$  does not truncate payloads it may be asked to send on recovery.

Assume FSOC the controller asks  $C$  for a diff  $B:t_B^*$ , except  $C$  has already compacted the diff. From the truncation algorithm,  $t_B^* \leq f(B)$ . Thus from the definition of  $f(B)$ ,

$t_B^* \leq f(B) \leq t_B$  in each of  $C$ 's max clocks, and each  $C$  will have already received  $B:t_B^*$  if  $C$  were an intended recipient. Let  $t_{B,min}$  be the minimum of these values for  $t_B$  and  $t_{B,max}$  the maximum. It follows that  $t_B^* \leq t_{B,min}$ . This contradicts the recovery algorithm, which requires  $t_{B,min} < t_B^* \leq t_{B,max}$  (§3.2.2). Thus  $C$  does not compact diffs it may be asked to send on recovery.

### 6.2.5 Summary

In summary,  $C_i$ 's inputs are valid as long as  $B'$  sends outputs that don't violate Regularity, while still satisfying Eventuality and Monotonicity. The messages that  $A_i$  send combined with the ordering defined by the diffs in  $C_i$  enable  $B'$  to determine an ordering of inputs to send outputs to its children. Log truncation does not affect the correctness of the recovery protocol. Any solution that satisfies the three execution replay constraints produces a valid execution, and there are many possible algorithms that can find a solution. ■

## 6.3 Proof: eventually, the system can recover again after a failure.

After recovering  $B$  as  $B'$ , the system will eventually be able to recover from the failure of another node because the recovery-related bookkeeping in all nodes will eventually be consistent with an execution that could have happened had no failure happened at all.

The protocol can recover from a failure of  $B'$ .  $B'$ 's state is lost on failure and does not affect the recovery of itself. Any bookkeeping state in  $A_i$  and  $C_i$  is consistent because the nodes continued operating from a previously-valid execution.

The protocol can recover from a failure of  $C_i$  once  $C_i$ 's children have updated their max clocks such that  $t_B \geq t_{B'}$  of  $B'$ 's max clock.  $t_{B'}$  is at least  $t_B^*$ , the root time of  $T^*$  from the recovery protocol after recovery, and  $B'$  can only resume sending messages after this time. Once the condition is met, the recovery protocol will then only ask  $B'$  to resume sending messages from  $t_B^* + 1$  or greater. The condition must eventually be met due to the periodic update mechanism for log truncation (§3.4.2).

The protocol can recover from a failure of  $A_i$  once  $B'$ 's siblings have updated their max clocks such that  $t_{A_i} \geq t_{A_i}^*$ , where  $t_{A_i}^*$  is the value of  $A_i$ 's time in  $T^*$  from the recovery protocol.  $B'$  can only send diffs of  $A_i$  after this time. Once the condition is met, the recovery protocol will then only ask  $B'$  to send diffs from  $t_{A_i} + 1$  or greater. The condition must eventually be met due to the periodic update mechanism for log truncation (§3.4.2).

The recovery protocol for all other failed nodes do not interact with  $A_i$ ,  $B$ , or  $C_i$ . Since these nodes are not in the neighborhoods of the failed nodes, the failed nodes are not affected by the previous failure. ■



# Chapter 7

## Extensions and Future Work

### 7.1 Stateful recovery

Being able to recover stateless nodes with tree clocks is already a big win, as the system can leave the state in downstream nodes untouched. In future work, we would like to extend the algorithm to stateful nodes using distributed snapshots [5] or replication [26, 24] to recover state.

Distributed snapshots enable us to rollback a failed stateful node to a previously-consistent state. Unlike existing implementations of distributed snapshots [5], we would like to avoid rolling back other nodes in the graph, which requires global coordination to ensure exactly-once semantics. Fortunately, tree clocks tell us exactly how to replay the message payloads in upstream nodes to bring the state in the restarted node up-to-date with its children.

An alternative to recovering materialized state is replication. Backup replication is expensive because it adds an extra RTT for each stateful node [24]. Chain replication adds only half the RTT, and avoids having to persist state like in snapshots [26]. However, replicas have to be on different machines, complicating how to efficiently partition nodes onto computers to distribute load.

Since the outputs of stateful nodes depend on the results of previous outputs, stateful nodes need to include the most recent diff from each parent that contributed to the output. Fortunately, the only stateful node with multiple parents is the *join* operator, which has exactly two parents. Thus including both parents in the diff does not significantly change the diff size.

### 7.2 Multiple concurrent failures

The protocol as it is can already handle the concurrent failures of multiple nodes, as long as the nodes are not adjacent to each other. The recovery-related metadata needed to recover a failed node is in the node's neighbors, and since none of these neighbors have failed, we are able to recover.

We believe the protocol can be extended to handle the concurrent failures of multiple adjacent nodes. Consider the failure of a connected subgraph with depth  $d$ . If the depth of the min and max clocks increase to  $d + 2$ , and the depth of a message diff to  $d + 1$ ,

then the recovery protocol can apply to the subgraph and its immediate neighbors, as if the subgraph were the failed node. The subgraph's children's tree clocks can determine where the subgraph's parents should resume sending messages, and their diffs can determine what order to process the received messages in.

To be able to handle the failure of any subset of nodes,  $d$  needs to be the depth of the entire dataflow graph. Since we expect graphs to be large due to sharding, it is a convenient property that  $d$  is unaffected by horizontal blowup. We can further optimize diffs by only forwarding parts of the lineage that are non-deterministic. For example, if a node has a single parent, we can deterministically replay its messages just from lineage about its parent's messages.

As seen in §5.1.1, the concept of a node in our protocol can include multiple dataflow nodes, as long as messages are serialized at the inputs and the outputs, and computation is deterministic. Thus in a sense, the concept of a node in our protocol is more a unit of a recovery than a single dataflow node. We would like to explore in future work how to implement this protocol as a more coarse-grained abstraction.

## 7.3 Other work

- **Tree clock no-copy:** Copying the payload into the payload log in the implementation adds significant overhead due to the size of message data (§5.2). Since the logs are stored in-memory, we can avoid this overhead by tracking a reference to the data instead of copying it.
- **Partial state:** In Noria, partially-stateful nodes drop a message if the data has not been queried by downstream nodes [16]. We may be able to treat the message like a filtered message, where the children of the message are a function the message output and the state. We must also handle Noria's eviction messages.
- **Dynamic dataflow graphs:** Tree clocks rely on the structure of the graph to be static in order to be accurate. The solution to this may be as simple as notifying affected nodes of the changed graph structure before completing a migration.

# Chapter 8

## Conclusion

This thesis presents a causal logging approach to fault tolerance for streaming dataflow systems that rolls back and replays the execution of only the failed node, without any global coordination. This approach piggybacks a small, constant-size tree clock onto each message, incurring low runtime overheads and encapsulating enough information to recover the system to a state that is indistinguishable from one that never failed at all. In future work, we plan to extend the protocol to stateful nodes and multiple concurrent failures.



# Bibliography

- [1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, Savannah, GA, November 2016. USENIX Association.
- [2] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, and Reuven Lax, Sam McVeety and Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, August 2013.
- [3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, August 2015.
- [4] Lorenzo Alvisi, Karan Bhatia, and Keith Marzullo. Causality tracking in causal message-logging protocols. *Distrib. Comput.*, 15(1):1–15, January 2002.
- [5] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering*, 38(4), December 2015.
- [6] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015.
- [7] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. *SIGPLAN Not.*, 45(6):363–375, June 2010.
- [8] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.
- [9] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

- [10] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39:11–16, 07 1991.
- [11] James Cheney, Laura Chiticariu, and Wang-chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1:379–474, 01 2009.
- [12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.
- [13] Elmootazbellah Elnozahy. Manetho: Fault tolerance in distributed systems using rollback-recovery and process replication. 01 1994.
- [14] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976.
- [15] Daniel Ford, Francois Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9<sup>th</sup> USENIX conference on Operating systems design and implementation (OSDI)*, pages 61–74, 2010.
- [16] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 213–231, Carlsbad, CA, October 2018. USENIX Association.
- [17] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, September 1991.
- [18] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2<sup>nd</sup> ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 59–72, March 2007.
- [19] Martin Kleppmann and Jay Kreps. Kafka, Samza and the Unix philosophy of distributed data. *IEEE Data Engineering Bulletin*, 38(4):4–14, December 2015.
- [20] E. A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991.
- [21] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of the 6<sup>th</sup> Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2013.
- [22] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24<sup>th</sup> ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, November 2013.

- [23] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1:222–238, 1981.
- [24] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 827–838, New York, NY, USA, 2004. ACM.
- [25] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, 08 1992.
- [26] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.
- [27] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, New York, NY, USA, 2017. ACM.
- [28] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: Fault tolerance off the critical path. In *Proceedings of Symposium on Operating Systems Principles*, SOSP '19, 2019.
- [29] Adam Warski. What does kafka's exactly-once processing really mean? Blog post, July 2017.
- [30] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 15–28, April 2012.
- [31] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–438, November 2013.