

# Scalable fault tolerance for high-performance streaming dataflow

Gina Yuan (Stanford)

Advisors: Jon Gjengset (MIT), Malte Schwarzkopf (Brown), Robert T. Morris (MIT), Eddie Kohler (Harvard)

**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. We introduce a causal logging approach to fault tolerance 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. We implement and evaluate the protocol on Noria, a streaming dataflow backend for read-heavy web applications, showing sub-second recovery times with 1.5 millisecond runtime overheads.

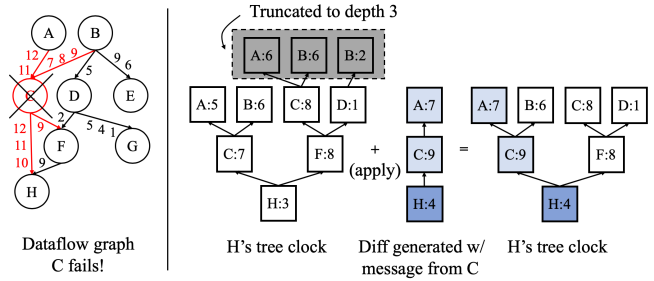
## 1 Research Problem and Motivation

Noria is a streaming dataflow system for read-heavy web application backends, intended to replace the classic MySQL / memcached stack [7]. In this use case, Noria must support thousands of latency-sensitive end users, sharding the backend to spread the work across multiple computers. This results in dataflow graphs with hundreds of nodes.

Noria distributes computation by assigning its dataflow nodes, which are relational operators, to computers. These nodes can either be stateful (e.g. join, aggregation) or stateless (e.g. filter, sharder, projection). Nodes communicate over one-way streams of messages along graph edges, and compute as a deterministic function of their inputs and state. Writes are eventually-consistent and observed exactly-once.

At the scale of a large website, machine failures are inevitable. Consider the failure of a computer with a single dataflow node (Fig. 1), and 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 [1, 5, 15].

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



**Figure 1.** Messages streaming through an example dataflow graph when *C* fails (left). *H* receives a message from *C*, then generates and applies a diff to its tree clock (right).

Noria redundantly purges and recomputes their states as well. Like in other coarse-grained lineage recovery solutions [16], recovery time with this protocol is proportional to the size of state in the graph. This is a problem for web applications, where massive amounts of data accumulate over time and high availability is mission-critical [6, 8].

An alternative is to restore the graph from a checkpoint without having to rebuild state. However, the global coordination required to either build or recover from checkpoints 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. We need a fault tolerance solution that can scale to a large number of shards without global coordination or slow offline recomputation.

## 2 Background and Related Work

Lineage-based recovery is a common fault tolerance technique for bulk-synchronous parallel (BSP) systems. When the lineage of a message is known before processing, these systems can rebuild lost state from partial results [4, 9, 17]. But while BSP systems have natural barriers for resuming computation due to their synchronous model of execution, as a continuous streaming system, Noria must rebuild state from scratch [7].

Checkpointing rolls back the system to a globally-consistent state after failure. Global checkpointing adds high runtime overheads due to the global coordination required for each checkpoint [10, 11]. Distributed and asynchronous checkpointing must roll back the entire graph to ensure exactly-once semantics, which is slow at scale [2, 3, 14].

Causal logging protocols piggyback message lineage onto each message so that on failure, the information on surviving nodes can be used to restore the system to a globally-consistent state [1, 5]. Asynchronous processing can remove the overhead of piggybacked lineage from the data path [15].

Our approach reduces the size of the lineage so that even synchronous processing incurs low runtime overheads.

### 3 Approach

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. We introduce a new abstraction called the *tree clock* that tracks the lineages of the messages each node has sent and received. Each node keeps a history of these messages in a *payload log* and the messages’ tree clocks in a *diff log*, along with a tree clock that reflects the node’s most recent lineage.

**Normal Operation.** A *tree clock* is an inverted tree of node IDs, each associated with an integer *time*. A path in the tree clock depicts a path a message could have taken through the dataflow. Each node  $N$  keeps a tree clock with root  $N$  and every possible path to  $N$  (Fig. 1):

- Initially, all times are 0.
- When  $N$  receives a message, copy the clock in the message, whose root is a parent node, and add  $N$  as a child with  $N$ ’s time+1. Call this clock a *diff*, and store the diff in the *diff log*. Apply the diff to  $N$ ’s clock by taking the greater value in corresponding entries.
- When  $N$  sends a message, include a copy of the last diff. Store the message in the *payload log*.

Note that this algorithm is local to  $N$  and requires no coordination with other nodes. By also truncating diffs based on the number of concurrent failures we want to be able to handle, the tree clock sent with each message is constant-size.

**Recovery Algorithm.** Consider a failed node  $B$  with multiple parents  $A_i$  and multiple children  $C_i$ . The system controller detects the failure and restarts  $B'$  on a computer:

1. Ask all  $C_i$  for their diff logs and tree clocks rooted at  $B$ . Let  $t_{B,min}$  be the minimum time for  $B$  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. Include  $T^*$  for  $B'$  to initialize its tree clock.
3. Tell each  $A_i$  to resume sending messages to  $B'$  at  $1 + A_i$ ’s time in  $T^*$ .

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. Since  $B'$  just started up, it has an empty payload log. It generates the next diffs to send using  $T^*$ , and filters sent duplicates based on where  $B'$  was told to send messages to  $C_i$ . 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.

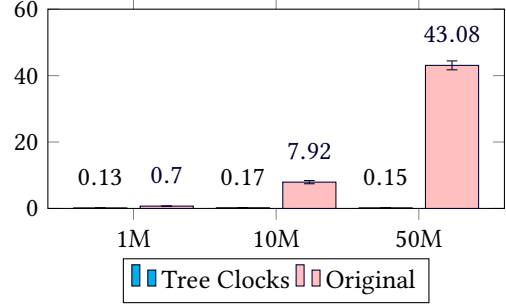


Figure 2. Number of articles versus recovery time (s).

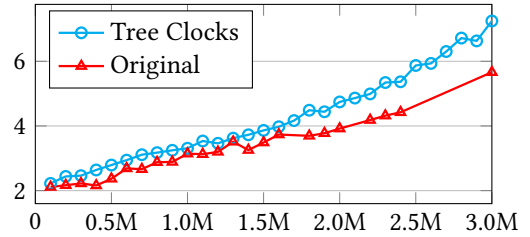


Figure 3. Load (ops/s) vs. write propagation time (ms).

**Execution Replay.** If  $A_i$  resends messages to  $B$  in a different order than that before failure,  $C_i$  may receive messages out-of-order or that are inconsistent with those of its siblings. The key insight is that the diff logs of  $C_i$  define a partial message ordering that can be used to generate a valid execution. The controller collects the diffs of root  $B$  with time greater than  $t_{min}$  and sends them to  $B'$ .  $B'$  then processes message from  $A_i$  in an order that can reproduce those diffs.

### 4 Results

We evaluate the performance of tree clocks and the recovery algorithm on a prototype implementation in Noria written in 4k lines of Rust compared to the original recovery algorithm.

We measure the recovery time from losing a *sharder* node that requires re-aggregating 50 million rows in each of 20 downstream shard when rebuilding state. In this simple experiment, tree clock recovery took 0.15s, a 290x improvement from lineage-based recovery, which took 43.08s (Fig. 2). Rebuilding state can only take longer as the system accumulates data over time.

We measure the overhead of tree clocks by comparing the write propagation times through the dataflow graph, which reflect read staleness. Near the maximum load of 3.0 million ops/s, the overhead of tree clocks is 28%, or 1.5ms (Fig. 3).

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 will extend the algorithm to stateful nodes using persistent, local snapshots [2] or replication [12, 13] to recover state. Fortunately, tree clocks already do most of the work, knowing where and in what order to resend messages.

## References

- [1] L. Alvisi, K. Bhatia, and K. Marzullo. Causality tracking in causal message-logging protocols. *Distrib. Comput.*, 15(1):1–15, Jan. 2002.
- [2] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering*, 38(4), Dec. 2015.
- [3] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.
- [4] J. Dean and S. 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.
- [5] E. Elnozahy. Manetho: Fault tolerance in distributed systems using rollback-recovery and process replication. 01 1994.
- [6] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, pages 61–74, 2010.
- [7] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. 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, Oct. 2018. USENIX Association.
- [8] J. Gray and D. P. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, Sept. 1991.
- [9] M. Isard, M. Budi, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 59–72, Mar. 2007.
- [10] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2013.
- [11] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, Nov. 2013.
- [12] M. A. Shah, J. M. Hellerstein, and E. 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.
- [13] R. van Renesse and F. 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.
- [14] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. 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.
- [15] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica. Lineage stash: Fault tolerance off the critical path. In *Proceedings of Symposium on Operating Systems Principles, SOSP '19*, 2019.
- [16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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, Apr. 2012.
- [17] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–438, Nov. 2013.