No-sync: coordination-free fault-tolerance in streaming

†‡Andrea Lattuada, Vasiliki Kalavri,
†Moritz Hoffmann*
Systems Group, Dep. of Comp. Sci., ETH Zürich
firstname.lastname@inf.ethz.ch

Frank McSherry
Unaffiliated
mcsherry@gmail.com

Introduction Streaming computations work on continually changing data streams, and fault tolerance is crucial to recover state corresponding to inputs that have since passed. Naive implementations require sophisticated fault tolerance, because they must record the behavior of otherwise unstructured computation. We find that by restricting our attention to functional computations mapped over streams of data, we can rely on much simpler and more efficient mechanisms.

Motivation Operator state in streaming jobs is very valuable and should be guarded against failure: lack of fault-tolerance would result in incorrect results after recovery. Additionally, streaming jobs run for long periods of time, accumulating state over several days or even months: reprocessing all input in the case of failures would be prohibitively expensive and time-consuming. At the same time, the common approaches to fault-tolerance based on snapshots or active replication provide additional guarantees that are often unnecessary and can have a significant negative impact on latency and overall performance.

We explore a technique that reduces the impact on steady-state performance by moving the replication out of the critical path, thanks to careful selection of the right guarantees.

Marker-based checkpointing vs. dependency tracking Marker-based checkpointing mechanisms (i.e. Flink[F]), require cross-operator coordination to provide consistency guarantees for nondeterministic computations. When in-band coordination markers are used, (i) the operators have to stall waiting for all markers to be aligned and (ii) cyclical topologies will deadlock.

Alternately, RDDs organize stage output in large chunks, which can be recomputed by inspecting their dependencies in the dataflow graph. This removes the need for explicit synchronization, but doesn't provide the opportunity for incremental checkpoints, and so reduces the frequency with which checkpoints can occur.

Our solution We adopt the logical boundaries of RDD approaches, but replace the full checkpoint with a log-structured merge-tree (LSM).

LSMs support incremental updates with low cost, recording only the changed entries in an append-only log. After each stage of a functional computations applied over streaming data, we record newly produced LSM layers, and perform maintenance (merging) as required. Layers can be persisted as resources permit, trading recovery time against run-time overhead.

Anticipated benefits of log-structured state The immutable and append-only nature of the log enables asynchronus replication of the state with minimal coordination. We don't require a consensus-based commit protocol: as soon as a batch of updates for a certain sequence number has been replicated to enough machines to ensure resilience to failure and network partitions, we can promote that batch to a safe rollback point and downgrade the persistence requirements for the inputs correspondingly.

This fundamentally changes fault-tolerance to a performance optimization, where different persistence levels (in memory, on disk, replicated remotely) can be used as resources permit. The more layers of the LSM can be persisted, at additional runtime cost, the more quickly one can recover from different classes of failures. When lightly loaded we can persist regularly to each level, ensuring brisk recovery; when more heavily loaded the persistence can be scaled back at the cost of greater potential recover times.

Recovery The recovery process is very simple in no-sync. Since all computations are functional and the log is append-only, the replicated data is always consistent and therefore we can always recover from the input. Further, if results are already available in the log, re-computation can be avoided. In practice, the recovery process starts new copies of the failed operators in the surviving nodes where their input collections have been replicated. The logs are rolled back to a safe point corresponding to a complete (logical) timestamp, and execution is resumed.

Other features The ability to selectively roll-back operators to any previous consistent state enables a variety of applications beyond fault-tolerance. A section of the computation can be rewound and then resumed with a tweaked version of the business logic to compare the outcomes. In a similar way, the replicated logs can serve as the basis for speculative execution of parallel tasks to mitigate the adverse effects of stragglers.

References

[RDD] M. Zaharia et al., *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.*

[F] P. Carbone et al., *State Management in Apache Flink Consistent Stateful Distributed Stream Processing.*

*†PhD student, ‡poster presenter

No-sync: coordination-free fault-tolerance in streaming

†Andrea Lattuada, †Vasiliki Kalavri, †Moritz Hoffmann, ‡Frank McSherry

MOTIVATION

- Any non-trivial **streaming** application needs to maintain **state**
  - rolling aggregation, windows
- Streaming applications run for days, months, even years
- Distributed systems *will* fail

EXISTING FAULT-TOLERANCE MECHANISMS

- Snapshot-based and transaction-based
  - cost: requires coordination
  - ensures consistency for non-deterministic computation (many computations are deterministic)
- Active replication to identical standbys
  - cost: requires 2x resources
  - provides zero-downtime guarantees (often not necessary)

PERFORMANCE TRADE-OFF

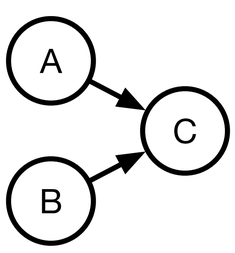
- Tradeoff between the steady-state failure-free overhead and the time it takes to recover
- We can reduce the failure-free overhead by focusing on the right guarantees

INCREMENTAL STATE UPDATES ARE PRACTICAL IN FUNCTIONAL COMPUTATIONS

LSM trees enable incremental updates with low cost

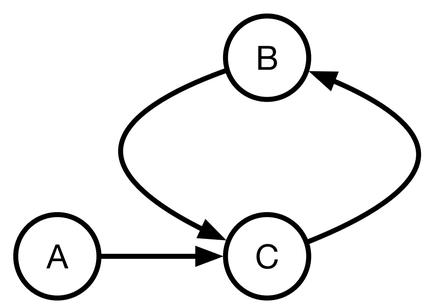
We can use them to cache recovery checkpoints for functional computations applied over streaming data

TRADITIONAL CHECKPOINTING: THE PROBLEM WITH COORDINATION



- Marker-based approach
  - waits for a checkpoint marker from each input
  - has to wait for all markers before proceeding

- Flink (RBEA)[1]: **Operator stall >500ms** ▶ directly affects latency

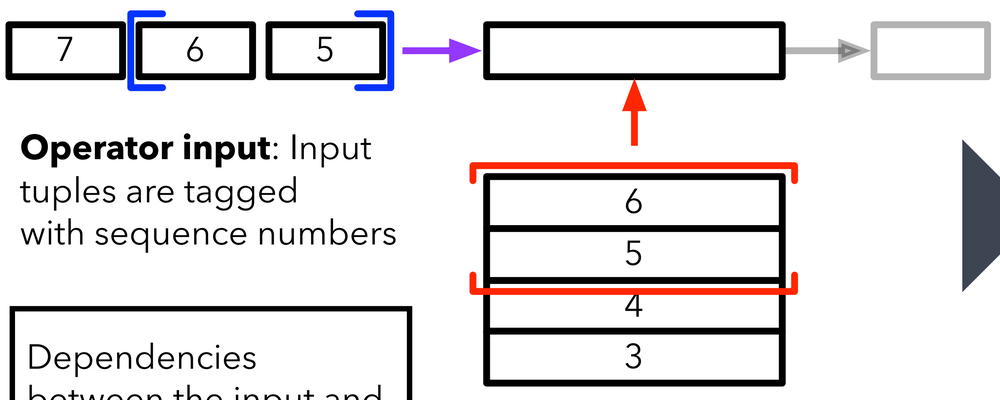


- Marker-based approach
  - waits for a checkpoint marker from each input
  - would deadlock in cyclical topologies

- Flink (RBEA)[1]: **100s for 100GB state snapshot**

OUR APPROACH: EXPLICIT DATA DEPENDENCIES AND INCREMENTAL STATE UPDATES

Dataflow edges imply data dependencies

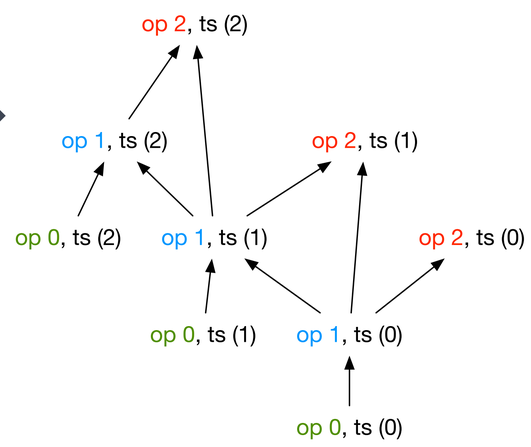


Operator input: Input tuples are tagged with sequence numbers

Dependencies between the input and the state updates.

Operator log: Incremental updates to the state are tagged with sequence no.

We derive a dependency graph between groups of tuples and state updates



Incremental updates can be performed **asynchronously**, with **no coordination**

- **Minimal** steady-state failure-free **overhead**
- **Minimal recomputation** on recovery by exploiting data dependencies

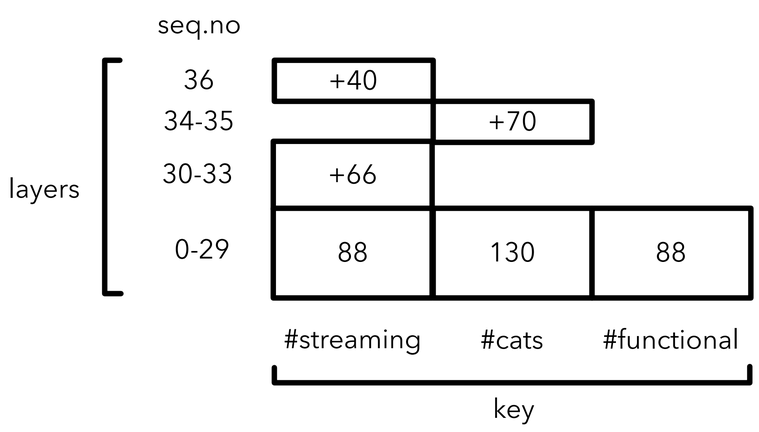
Expected perf benefits

- No operator stalls (>500ms lower latency than Flink with 200G state)
- ms-scale incremental updates

LOG STRUCTURE

The in-memory operator state is an indexed view of the incremental updates.

An LSM-like data structure stores the state updates as small incremental updates at the top. We merge layers much like an LSM, trading off recovery time and run-time overhead.



RECOVERY

- functional computation, can always recover from input (via rollback)
- replicated data always consistent (append only)
- re-computation can be avoided if results are available in log

- low recovery cost makes a short failure detection timeout feasible (potentially ~10s faster then Flink)

FUTURE WORK

- time travel
- dynamic rescaling

[1] P. Carbone, S. Ewen, G. Ora, S. Haridi, S. Richter, and K. Tzoumas, "State Management in Apache Flink"

†Systems Group, Department of Computer Science, ETH Zurich
firstname.lastname@inf.ethz.ch

‡Unaffiliated
mcsherry@gmail.com