Lineage Stash: Fault Tolerance Off the Critical Path

Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, Ion Stoica
Low latency is increasingly important in data processing systems.

Data processing is used today in online systems.

- Stream processing
- Graph processing
- Control systems

Data processing at large scale also requires the ability to recover results after a failure.
Tradeoff between low latency and recovery time

Recovery overhead

Runtime overhead

× Lineage stash
Stream processing example

Display the number of times “UCB” has been queried.

```json
{
    user_id: ..., 
    keyword: "UCB",
}
```

10s of milliseconds
Stream processing example

```
{
  user_id: ...
  keyword: {
    "Cal": user_id: ...
    "Beat Stanford": keyword:
      "Go bears",
  }
}
```

Task: A few to 10s of milliseconds

-Time-
Stream processing example

- Time

Select 1
Select 2
Filter
Counter
Stream processing example
Tradeoff between low latency and recovery time

Global checkpointing
Global checkpointing

- Select 1
- Select 2
- Filter
- Counter

["UCB", "Bears"]
["UCB", "UCB"]

+2
+1
Global checkpointing

Select 1

Select 2

Filter

Counter

Roll back **failed** process

[“UCB”, “Bears”]

[“UCB”, “UCB”]

-Time-
Global checkpointing

Select 1
Select 2
Filter
Counter

Roll back **upstream** processes

[“UCB”,
 “Bears”]
Global checkpointing
Global checkpointing

Select 1
Select 2
Filter
Counter

-Time-

[“UCB”, “Bears”]
[“UCB”, “UCB”]

+1
+2
Global checkpointing

Select 1
Select 2
Filter
Counter

[“UCB”, “Bears”]
[“UCB”, “UCB”]

Must also roll back downstream processes

Time
Global checkpointing

Select 1

Select 2

Filter

Counter

-Time→

["UCB", "Bears"]

["UCB", "UCB"]
Global checkpointing
Global checkpointing

Take a global checkpoint on some interval and do a global rollback on any failure

**Low runtime overhead**

**High recovery overhead**
Tradeoff between low latency and recovery time

- Recovery overhead
- Runtime overhead
- Global checkpointing
- Logging
Logging

Select 1

Select 2

Filter

Counter

- Time -

["UCB", "Bears"]

["UCB", "UCB"]

+2

+1
Logging

Record messages

[“UCB”, “Bears”]

[“UCB”, “UCB”]

+2

+1

Select 1

Select 2

Filter

Counter
Logging

Record nondeterministic execution order
Logging

Select 1

Select 2

Filter

Counter

- Time 

Logging

[“UCB”, “Bears”]

[“UCB”, “UCB”]

+2

+1

---

22
Logging

Select 1

Select 2

Filter

Counter

-Time→

[“UCB”, “Bears”]

[“UCB”, “UCB”]

+2

+1
Logging

- Select 1
- Select 2
- Filter
- Counter

["UCB", "Bears"]
["UCB", "UCB"]
+2
+1
Logging

Select 1

Select 2

Filter

Counter

-Time-

["UCB", "Bears"]

["UCB", "UCB"]

+2

+1
Logging

Record additional information between checkpoints so that only failed processes need to be rolled back

Low recovery overhead
High runtime overhead
Log the **lineage** to reduce the **amount** logged

Select 1

Select 2

Filter

Counter

-Time–

[“UCB”, “Bears”]

[“UCB”, “UCB”]

+2

+1

Log the **lineage** to reduce the **amount** logged
Log the **lineage** to reduce the **amount** logged

Select 1

Select 2

Filter

Counter

-Time

Pointers to data

[“UCB”, “Bears”]

[“UCB”, “UCB”]

+2

+1

Log the **lineage** to reduce the **amount** logged!
Log the **lineage** to reduce the **amount logged**

Task descriptions

- [“UCB”, “Bears”]
- [“UCB”, “UCB”]

Pointers to data
But logging still requires a **synchronous** round-trip to remote storage.
But logging still requires a **synchronous** round-trip to remote storage
But logging still requires a synchronous round-trip to remote storage
But logging still requires a synchronous round-trip to remote storage
But logging still requires a **synchronous** round-trip to remote storage

Scheduling delay / task $\geq 1\text{RTT} + 1\text{RPC}$

**Task latency depends on global storage latency**
Lineage stash contribution

How do we achieve both low runtime and low recovery overhead for fine-grained data processing applications?

Solution: Asynchronously log the lineage off the critical path of execution.

Lineage reconstruction to reduce amount logged

Causal logging to log nondeterminism
Lineage stash architecture

Persistent key-value store.

Replicated for durability.

Sharded for horizontal scalability

Global Lineage Storage
Lineage stash architecture

Select 1  Select 2  Filter  Counter

Global Lineage Storage
Lineage stash architecture

Local, volatile cache for application objects.
Lineage stash architecture

Local, volatile cache for lineage.
Lineage stash: Logging the lineage, asynchronously

(1) Write lineage to **local, volatile** lineage stash.

(2) **Asynchronously** flush to **remote** storage.

Scheduling delay / task = \(1RTT + 1RPC\)

Task latency **independent of** global storage latency
Lineage stash: Logging the lineage, asynchronously

1. Write lineage to **local, volatile** lineage stash.
2. **Asynchronously** flush to **remote** storage.
   
   Scheduling delay / task = $1_{RTT} + 1_{RPC}$

   Task latency **independent of** global storage latency
Lineage stash: Logging the lineage, asynchronously

(1) Write lineage to local, volatile lineage stash.

(2) Asynchronously flush to remote storage.

Scheduling delay / task = $1RTT + 1RPC$

Task latency independent of global storage latency
Lineage stash: Logging the lineage, asynchronously

[“Go bears”, “Beat Stanford”, “Cal”]

Filter

Stash

Counter

Stash

Global Lineage Storage
Lineage stash: Logging the lineage, asynchronously.
Lineage stash: Logging the lineage, asynchronously
Lineage stash: Logging the lineage, asynchronously

[“Go bears”
“Beat Stanford”
“Cal”]

[“UCB”, “UCB”]
Lineage stash: Logging the lineage, asynchronously

[“Go bears”, “Beat Stanford”, “Cal”]

[“UCB”, “UCB”]
Lineage stash: Logging the lineage, asynchronously

[“Go bears”, “Beat Stanford”, “Cal”]

Filter

Stash

Counter

Stash

Global Lineage Storage

[“UCB”, “UCB”]
Lineage stash: Logging the lineage, asynchronously

[“Go bears”, “Beat Stanford”, “Cal”]

[“UCB”, “UCB”]
Lineage stash: Logging the lineage, asynchronously

(3) Forward uncommitted lineage.

[“Go bears”,
“Beat S”,
“Cal”]

Filter

Stash

Global Lineage Storage

Uncommitted Lineage

Counter

Stash

Counter
Lineage stash: Logging the lineage, asynchronously

[“Go bears”, “Beat Stanford”, “Cal”]

(3) Forward uncommitted lineage.
Lineage stash recovery

Diagram showing the connections between Filter, Counter, and Global Lineage Storage.
Lineage stash recovery
Lineage stash recovery
Lineage stash recovery
Lineage stash recovery

(1) Flush uncommitted lineage
Lineage stash recovery

(2) Ack recovering process

Global Lineage Storage
Lineage stash recovery

(3) Retrieve and replay lineage
Tradeoff between low latency and recovery time

![Diagram showing tradeoff between recovery overhead and runtime overhead with points for Global checkpointing, Lineage stash, and Logging marked with Xs.]
Latency without failures

Streaming wordcount:
- 32 m5.xlarge nodes
- 30s checkpoint interval

CDF

Latency (ms)

Better
Latency without failures

CDF

Global checkpointing

Better

Latency (ms)

Flink
Latency without failures

Logging the lineage **synchronously**
Latency without failures

Logging the lineage asynchronously!
Throughput during recovery

- Streaming wordcount:
  - 32 m5.xlarge nodes
  - 30s checkpoint interval

Throughput (100k records/s)

Time since start (s)
Throughput during recovery

- Throughput (100k records/s)
- Time since start (s)
- Global checkpoint
- Throughput during recovery
- Streaming wordcount:
  - 32 m5.xlarge nodes
  - 30s checkpoint interval
- Flink
- WriteFirst
- Lineage stash

Global checkpoint
Throughput during recovery

- 32 m5.xlarge nodes
- 30s checkpoint interval

Streaming wordcount:

Node failure

Checkpoint

Throughput (100k records/s)

Time since start (s)
Throughput during recovery

Global rollback and replay

Throughput (100k records/s)

Time since start (s)

Streaming wordcount:
- 32 m5.xlarge nodes
- 30s checkpoint interval

- Flink
- WriteFirst
- Lineage stash
- stash
Throughput during recovery

Throughput (100k records/s)

Time since start (s)

Checkpoint

Process new records since failure

Flink
WriteFirst
Lineage
stash

69
Throughput during recovery

![Graph showing throughput during recovery with various labels and annotations including Throughput (100k records/s), Time since start (s), Partial rollback, Checkpoint, Flink, WriteFirst, Lineage, stash.]
Latency during recovery

Streaming wordcount:
- 32 m5.xlarge nodes
- 30s checkpoint interval
Lineage Stash

See the paper (or email me: swang@berkeley.edu) for:

• Discussion and evaluation of other applications
• Nondeterminism in data processing applications
• Protocols for flushing and recovering the stash

Key idea: **Asynchronously** log the **lineage** and forward **uncommitted** lineage to guarantee recovery correctness.

Low latency during execution and low downtime after a failure for large-scale decentralized data processing applications.