FASTER State Management for Timely Dataflow

Matthew Brookes

Systems Group
Dept. Computer Science
ETH Zurich - Switzerland

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Problem Statement

Problem Statement

- Long-running streaming computations accrue large amounts of state

Problem Statement

- Long-running streaming computations accrue large amounts of state
- As state grows stream processing systems can:
  - Scale-out to distributed machines
  - Offload state to secondary storage

Problem Statement

- Long-running streaming computations accrue large amounts of state
- As state grows stream processing systems can:
  - Scale-out to distributed machines
  - Offload state to secondary storage
- We explore the trade-off between these approaches through an integration of Timely Dataflow with FASTER

Demonstration

Demonstration Outline

[1] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. NEXMark—A Benchmark for Queries over Data Streams DRAFT

Demonstration Outline

- NEXMark [1] Query 3 is an incremental join between two streams

[1] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. NEXMark—A Benchmark for Queries over Data Streams DRAFT

Demonstration Outline

- NEXMark [1] Query 3 is an incremental join between two streams
- Operator maintains the *people* and *auctions* relations

[1] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2002. NEXMark—A Benchmark for Queries over Data Streams DRAFT

Demonstration Outline

- NEXMark [1] Query 3 is an incremental join between two streams
- Operator maintains the *people* and *auctions* relations

```
SELECT (P.name, P.city, P.state, A.id)
FROM Auction A, Person P
WHERE A.seller = P.id
        AND (P.state IN (`OR’, `ID’, `CA’))
        AND A.category = 10;
```

[1] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier.
2002. NEXMark—A Benchmark for Queries over Data Streams
DRAFT

Timely Dataflow [2]

[2] https://github.com/TimelyDataflow/timely-dataflow
[3] Derek G Murray, Frank McSherry, Rebecca Isaacs, et al. 2013.
Naiad: a timely dataflow system

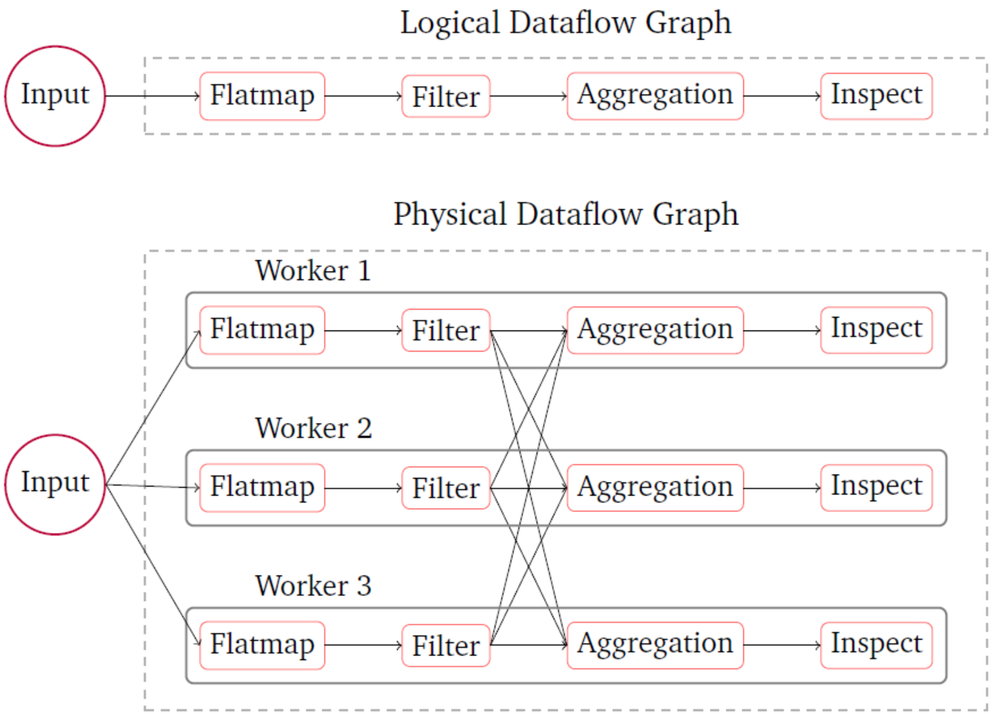
Timely Dataflow [2]

- A stream processing system based upon the Naiad [3] dataflow model

[2] https://github.com/TimelyDataflow/timely-dataflow
[3] Derek G Murray, Frank McSherry, Rebecca Isaacs, et al. 2013. Naiad: a timely dataflow system

Timely Dataflow [2]

- A stream processing system based upon the Naiad [3] dataflow model



[2] https://github.com/TimelyDataflow/timely-dataflow
[3] Derek G Murray, Frank McSherry, Rebecca Isaacs, et al. 2013. Naiad: a timely dataflow system

FASTER [4]

[4] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, et al. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates

FASTER [4]

- Hybrid-log structure spanning main memory and disk

[4] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, et al. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates

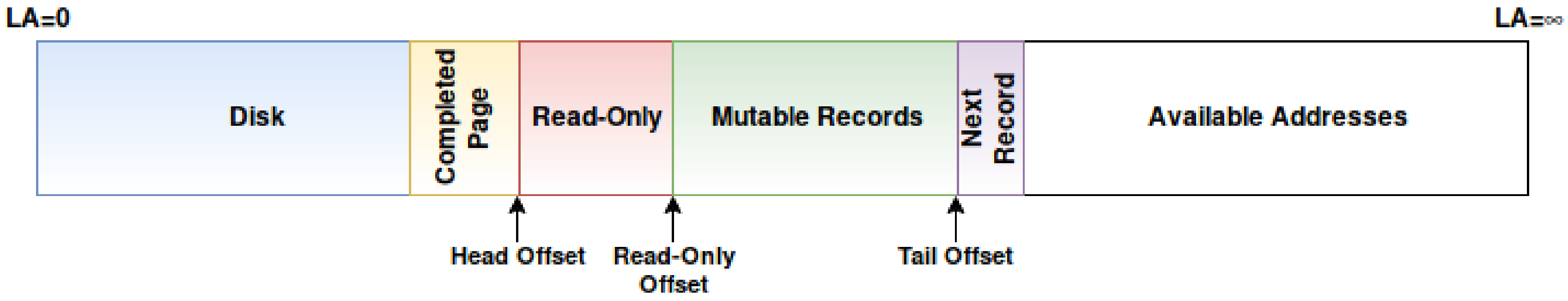
FASTER [4]

- Hybrid-log structure spanning main memory and disk
- In-memory *hot* and on-disk *cold* set

[4] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, et al. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates

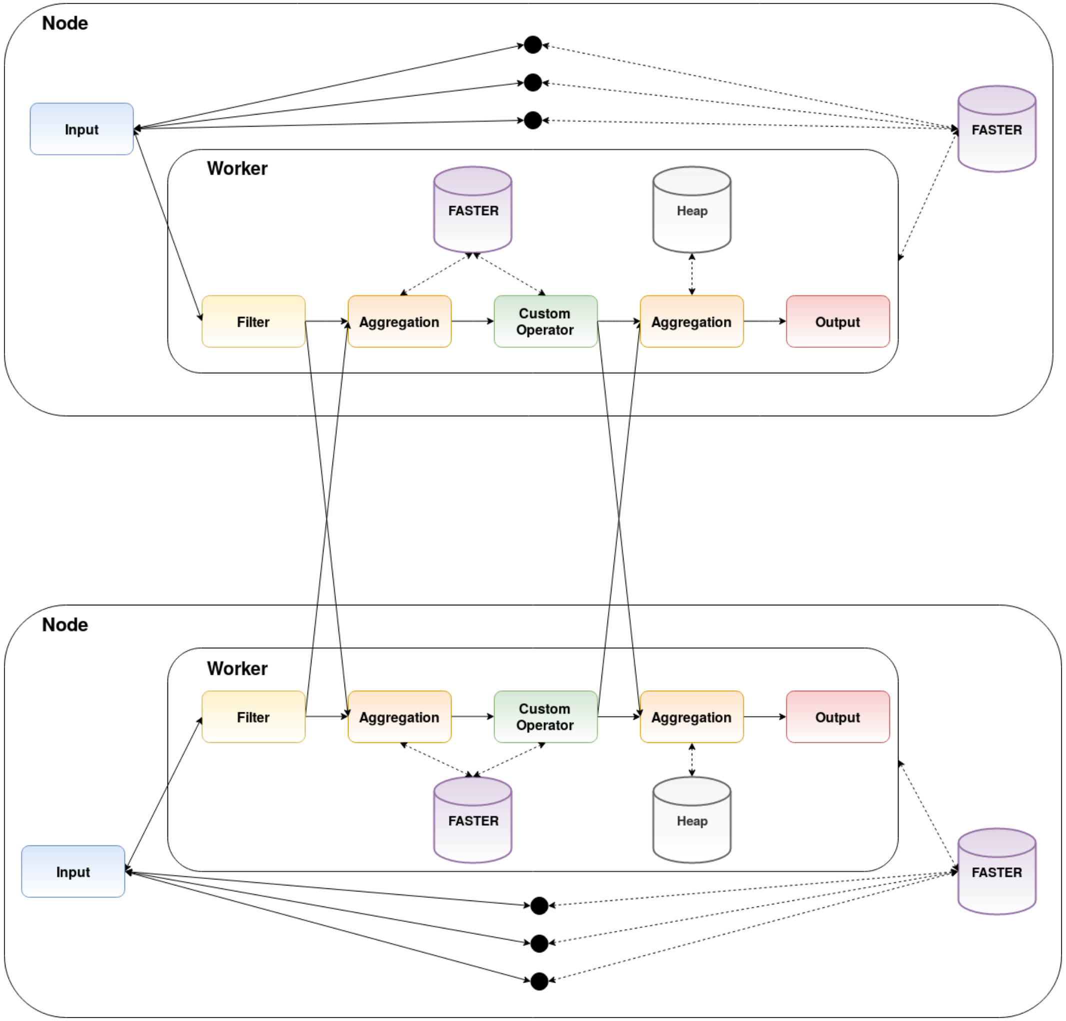
FASTER [4]

- Hybrid-log structure spanning main memory and disk
- In-memory *hot* and on-disk *cold* set

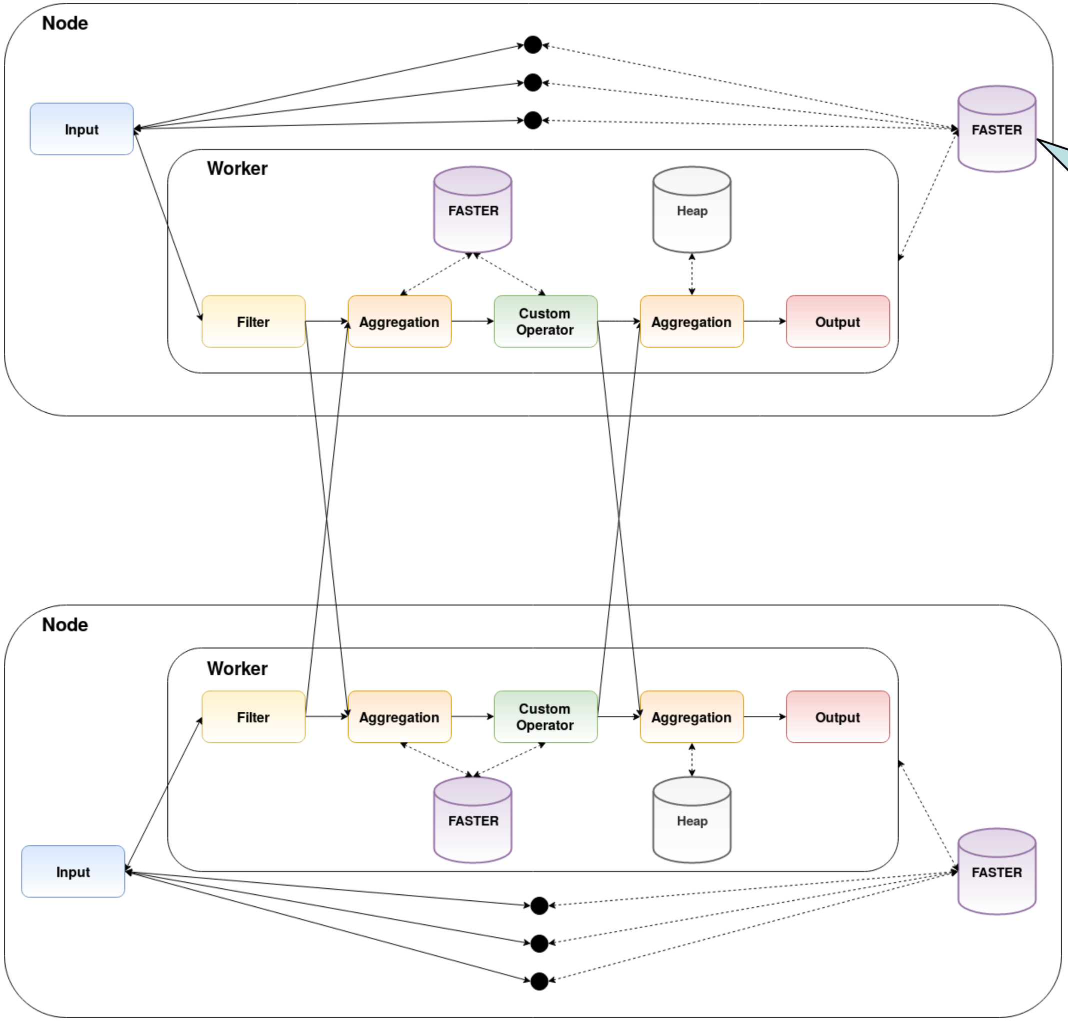


[4] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, et al. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates

FASTER State Management for Timely Dataflow

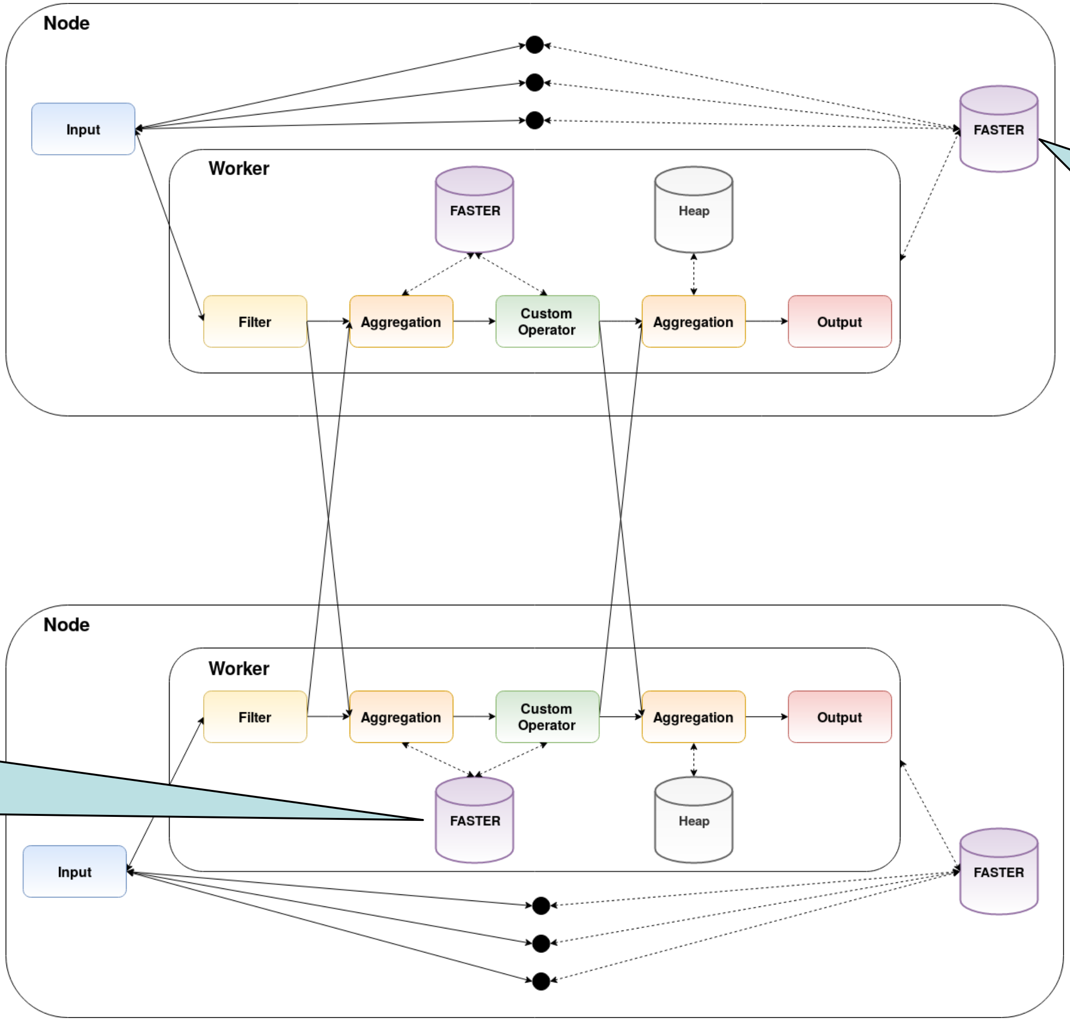


FASTER State Management for Timely Dataflow



Node-level backend

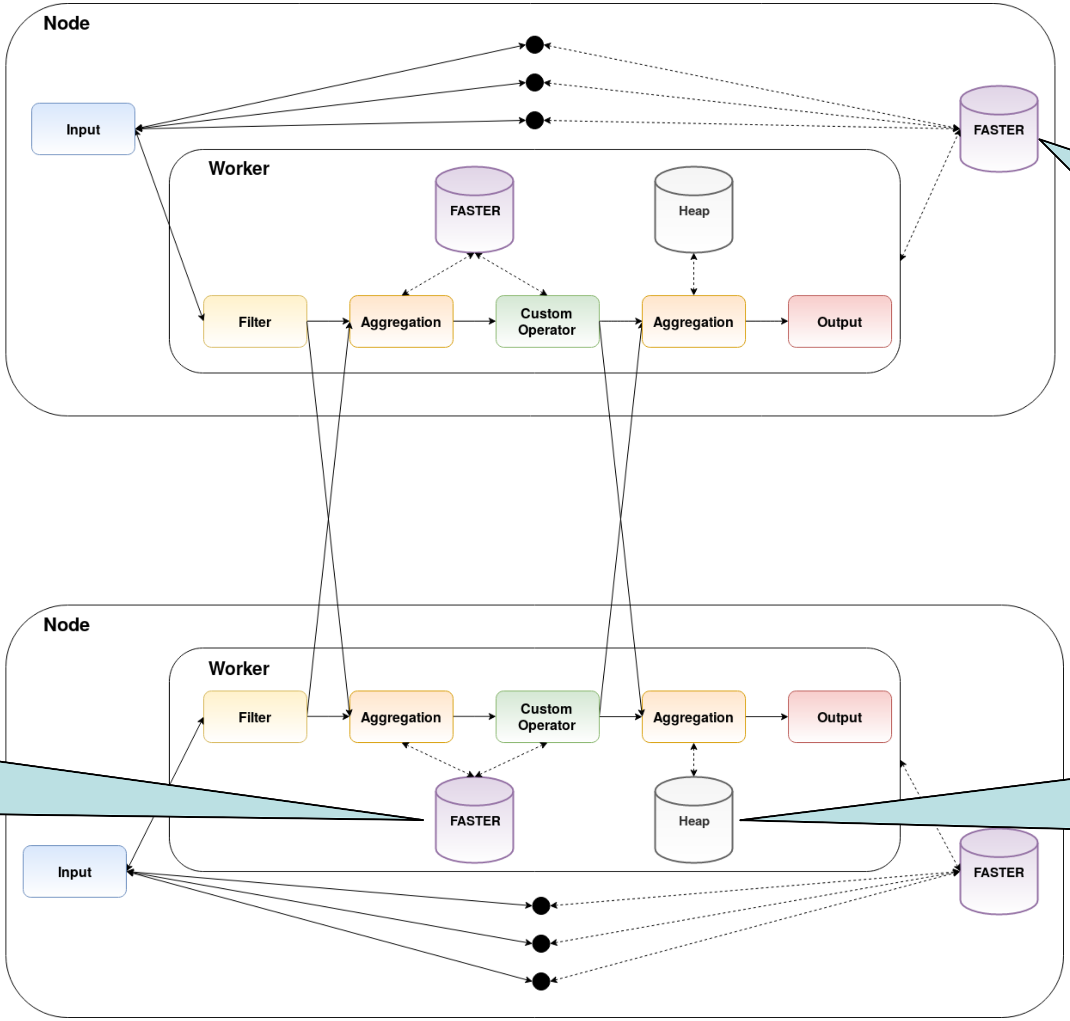
FASTER State Management for Timely Dataflow



Node-level backend

Worker-level backend

FASTER State Management for Timely Dataflow

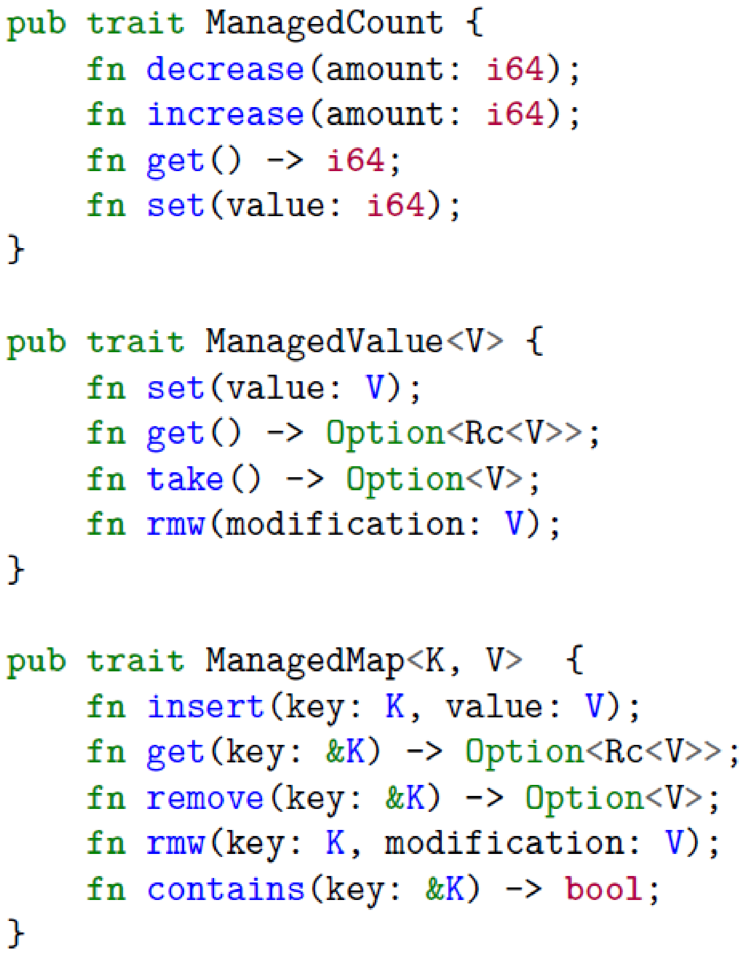


Node-level backend

Worker-level backend

Operator-level backend

Managed State Primitives

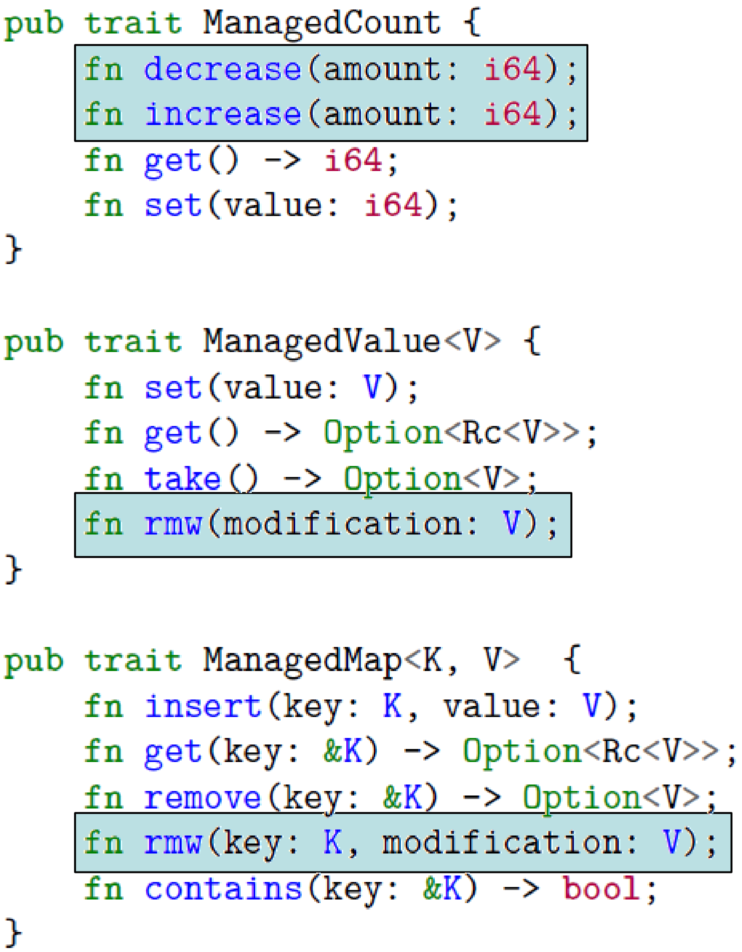


```
pub trait ManagedCount {
    fn decrease(amount: i64);
    fn increase(amount: i64);
    fn get() -> i64;
    fn set(value: i64);
}

pub trait ManagedValue<V> {
    fn set(value: V);
    fn get() -> Option<Rc<V>>;
    fn take() -> Option<V>;
    fn rmw(modification: V);
}

pub trait ManagedMap<K, V>  {
    fn insert(key: K, value: V);
    fn get(key: &K) -> Option<Rc<V>>;
    fn remove(key: &K) -> Option<V>;
    fn rmw(key: K, modification: V);
    fn contains(key: &K) -> bool;
}
```

Managed State Primitives



```
pub trait ManagedCount {
    fn decrease(amount: i64);
    fn increase(amount: i64);
    fn get() -> i64;
    fn set(value: i64);
}

pub trait ManagedValue<V> {
    fn set(value: V);
    fn get() -> Option<Rc<V>>;
    fn take() -> Option<V>;
    fn rmw(modification: V);
}

pub trait ManagedMap<K, V>  {
    fn insert(key: K, value: V);
    fn get(key: &K) -> Option<Rc<V>>;
    fn remove(key: &K) -> Option<V>;
    fn rmw(key: K, modification: V);
    fn contains(key: &K) -> bool;
}
```

Back to the demonstration

Evaluation

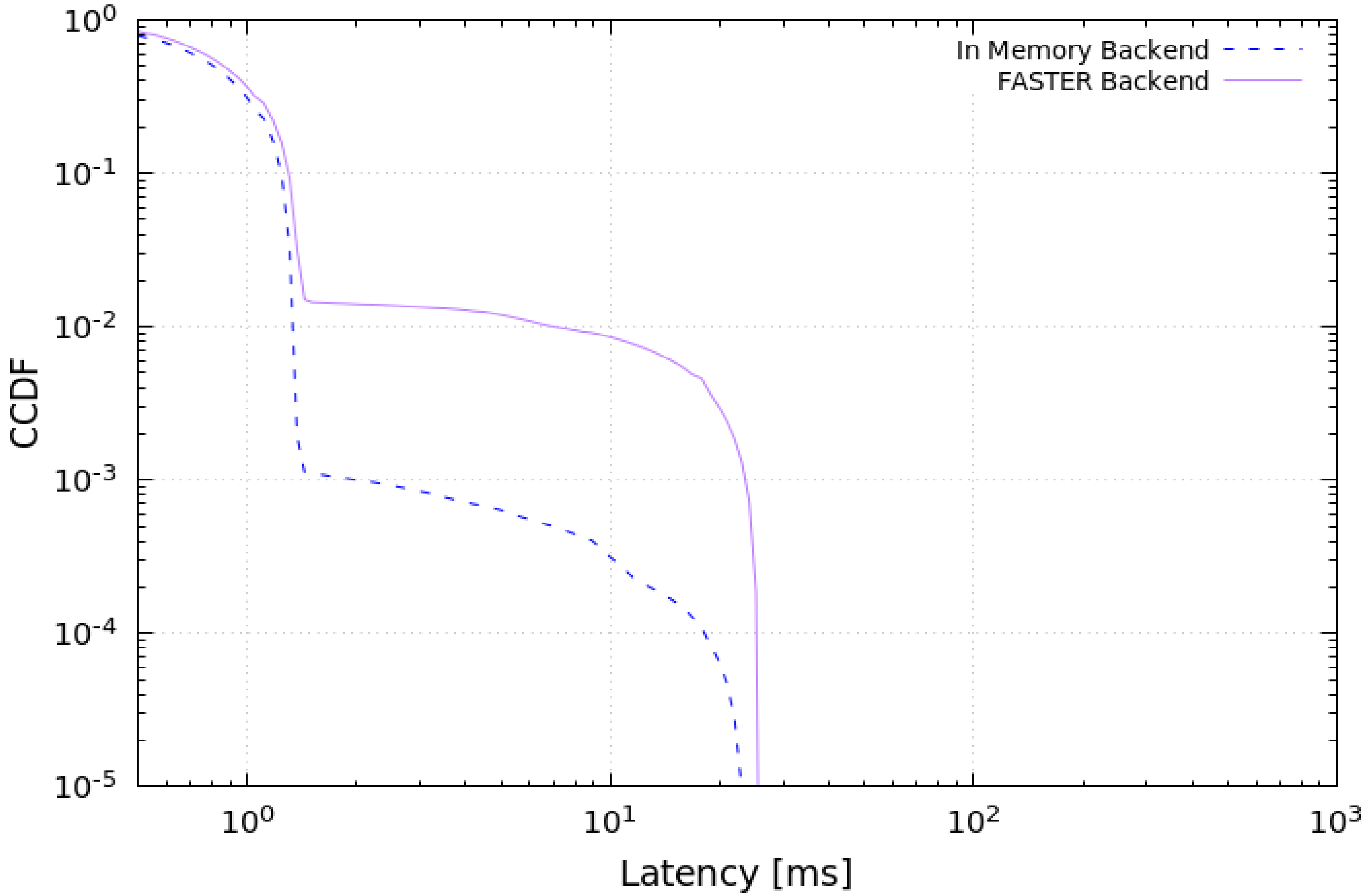
Evaluating FASTER State Management for Timely Dataflow

- Multiple machines with 4 physical cores and 16 GB RAM
- 200 GB non-volatile memory express (NVMe) SSD
- Machines communicate with up to 3500 Mbps bandwidth

NEXMark Query 3 recap

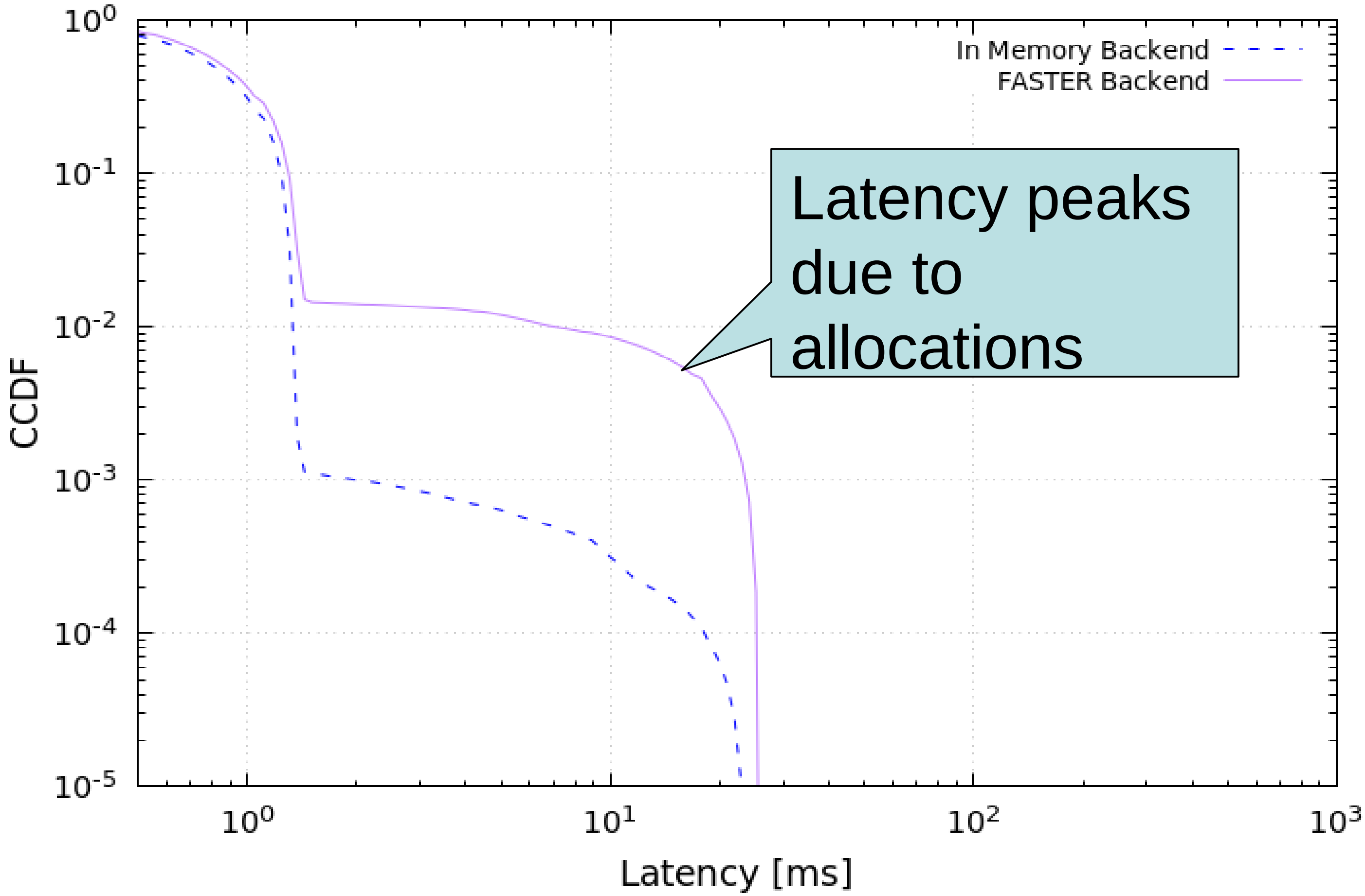
```
SELECT (P.name, P.city, P.state, A.id)
FROM Auction A, Person P
WHERE A.seller = P.id
        AND (P.state IN (`OR’, `ID’, `CA’))
        AND A.category = 10;
```

FASTER performance overhead versus native data structures



Query 3
(Incremental Join)

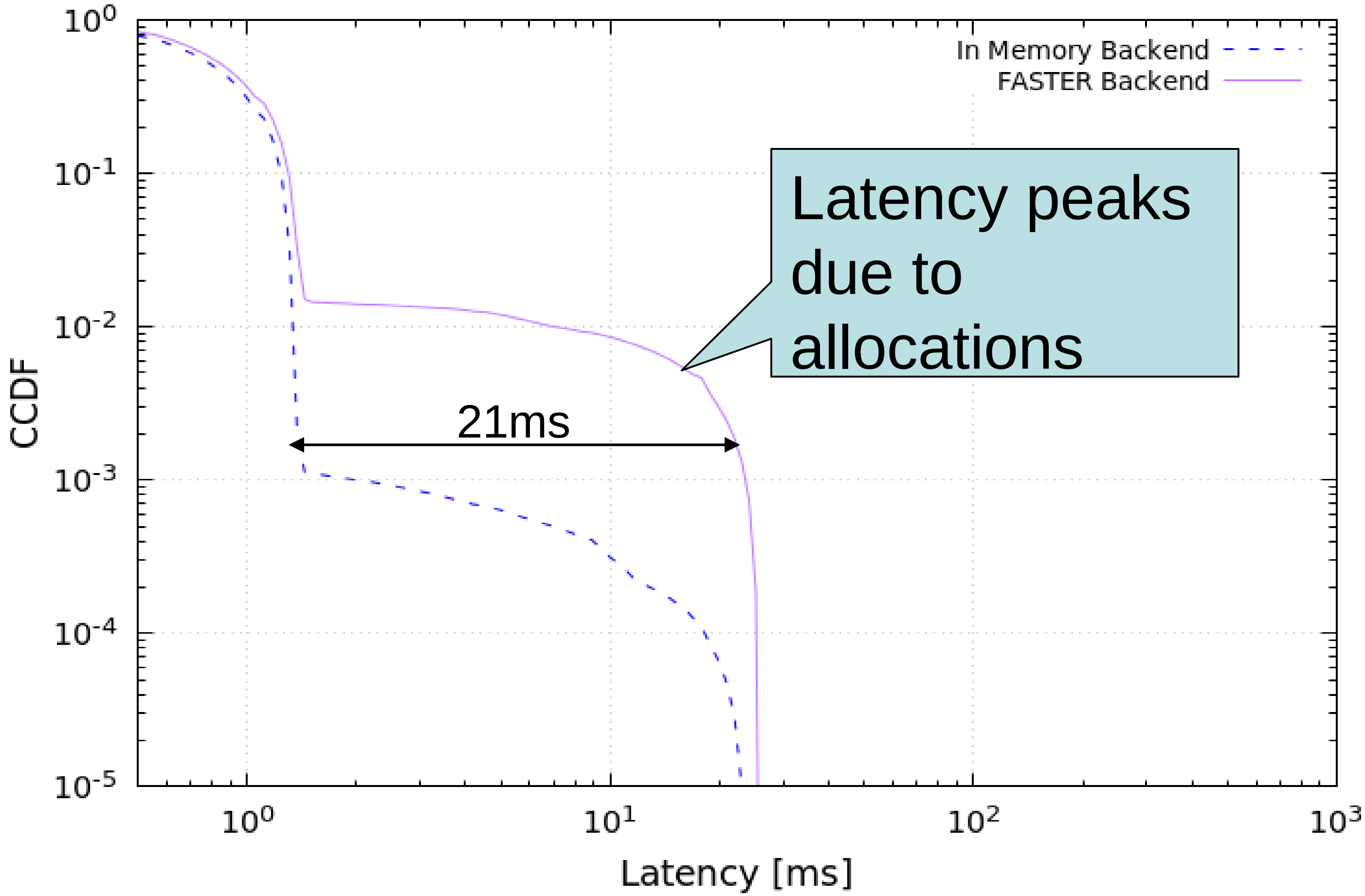
FASTER performance overhead versus native data structures



Latency peaks due to allocations

Query 3
(Incremental Join)

FASTER performance overhead versus native data structures



21ms

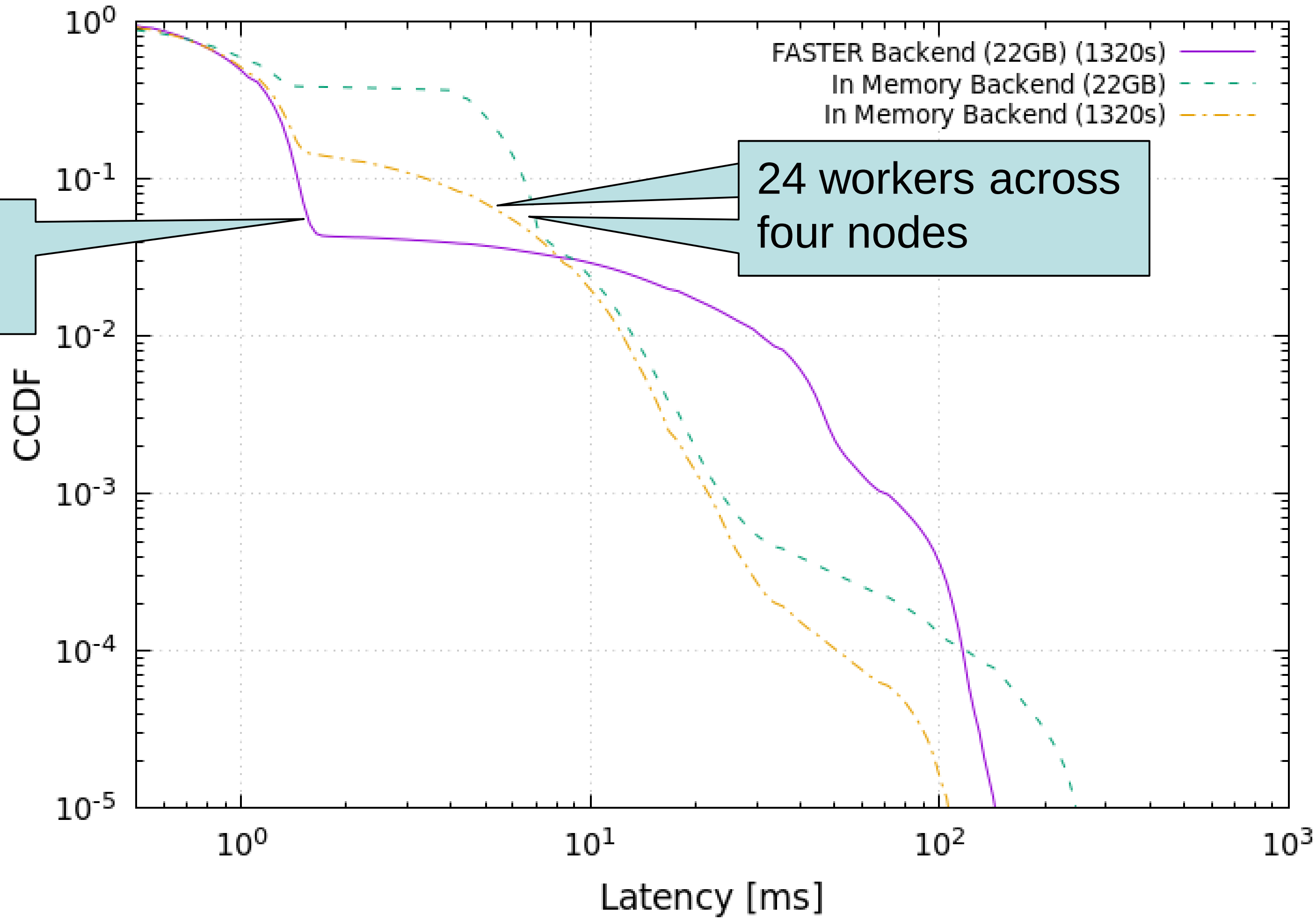
Latency peaks due to allocations

Query 3
(Incremental Join)

Evaluating the trade-off between scaling-out and using secondary storage

- Compare using FASTER on a single worker to scaling-out and using 24 workers across 4 nodes
- Run Query 3 for 1320s to accumulate 22 GB on FASTER
- Run in-memory for 1320s and 4983s to compare

Evaluating the trade-off between scaling-out and using secondary storage

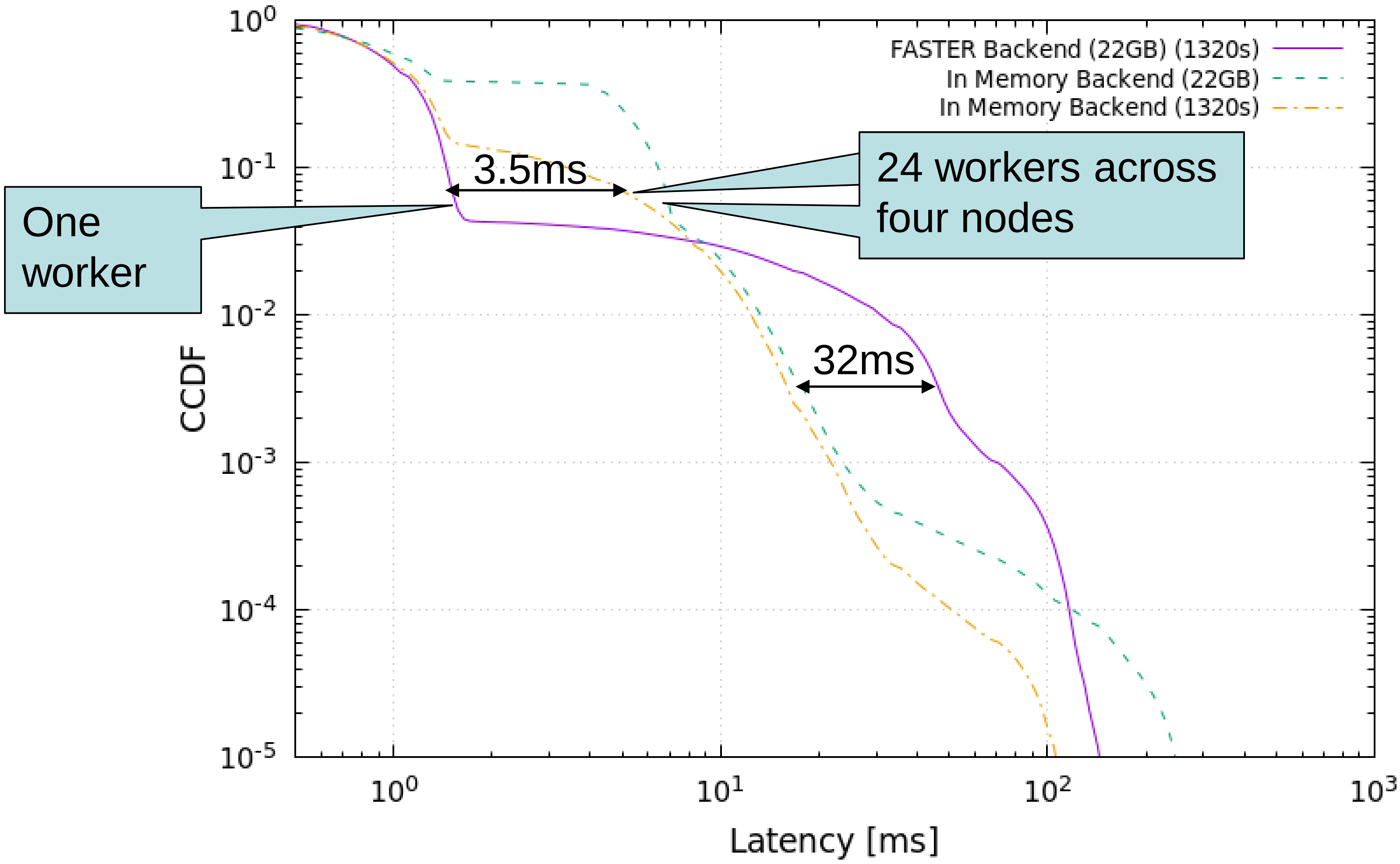


One worker

24 workers across four nodes

Evaluating the trade-off between scaling-out and using secondary storage

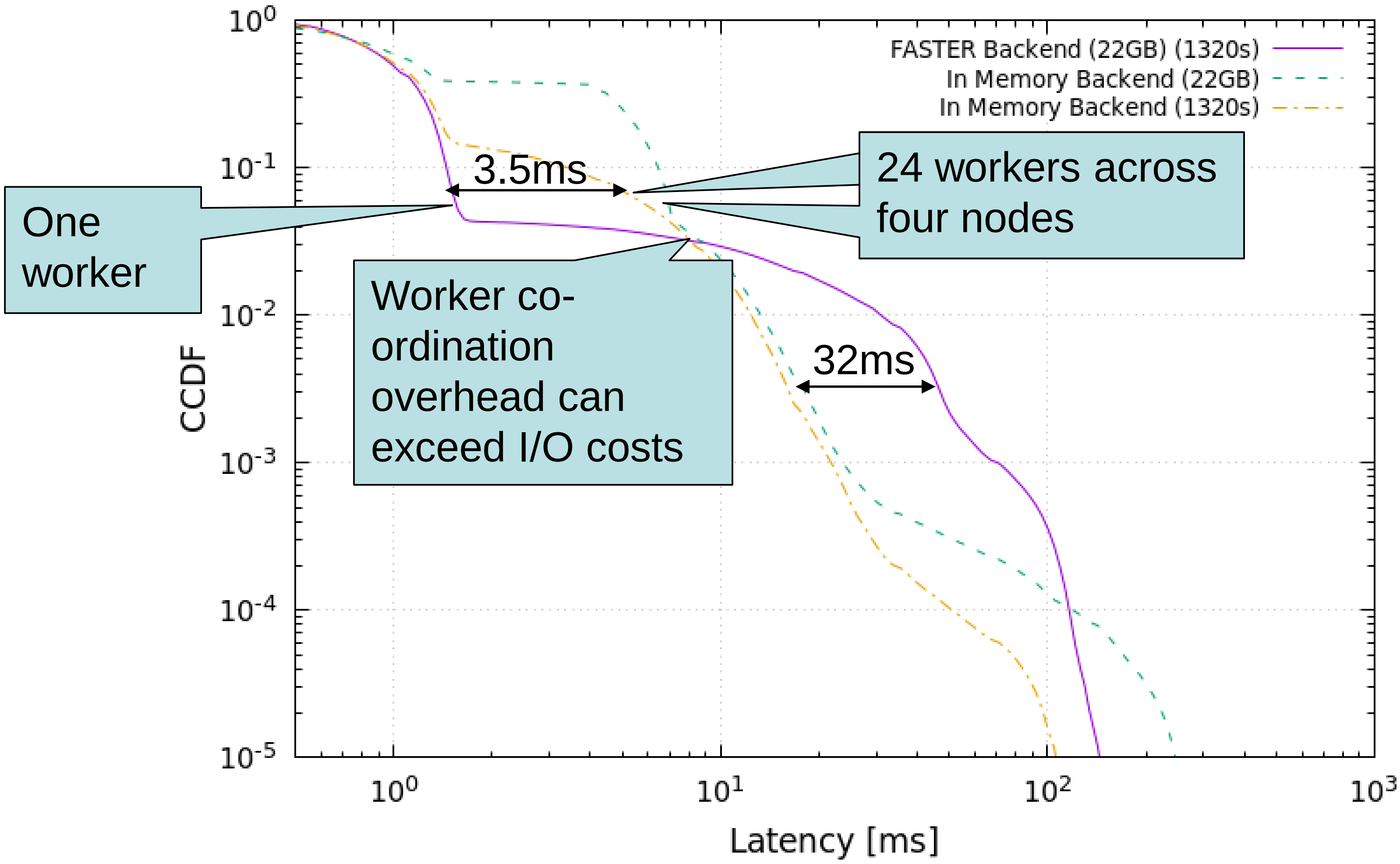
Systems@ETH zürich



One worker

24 workers across four nodes

Evaluating the trade-off between scaling-out and using secondary storage



One worker

24 workers across four nodes

Worker co-ordination overhead can exceed I/O costs

Conclusion

Conclusion

- State management for larger-than-memory operator state is a relevant topic

Conclusion

- State management for larger-than-memory operator state is a relevant topic
- Using FASTER to store state across main memory and secondary storage incurs acceptable overhead

Conclusion

- State management for larger-than-memory operator state is a relevant topic
- Using FASTER to store state across main memory and secondary storage incurs acceptable overhead
- In some cases it is preferable to rely on secondary storage for storing larger-than-memory state rather than scaling out to more nodes

Additional resources

- https://github.com/faster-rs/faster-rs
- https://github.com/faster-rs/FASTER
- https://github.com/matthewbrookes/timely-dataflow/tree/state_crate
- https://github.com/matthewbrookes/nexmark_timely_faster
- Matthew Brookes. 2019. *FASTER State Management for Strymon*. Master’s thesis. ETH Zürich.