IN SUPPORT OF WORKLOAD-AWARE STREAMING STATE MANAGEMENT

Vasiliki Kalavri
vkalavri@bu.edu

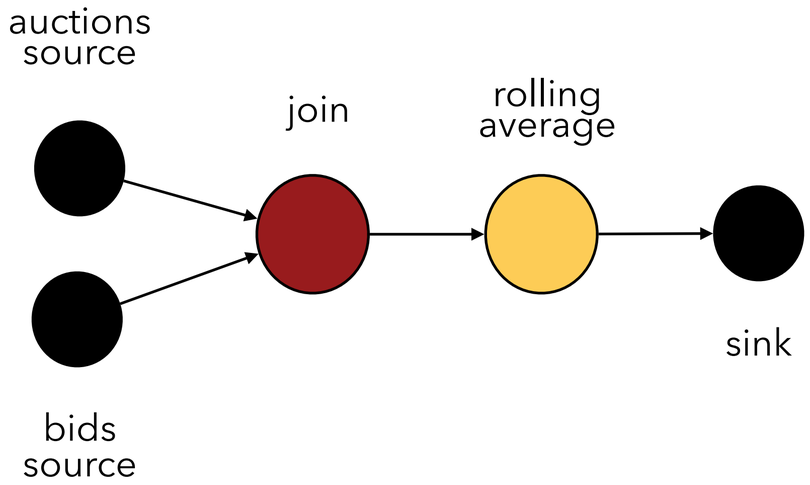
John Liagouris
liagos@bu.edu

HotStorage 2020

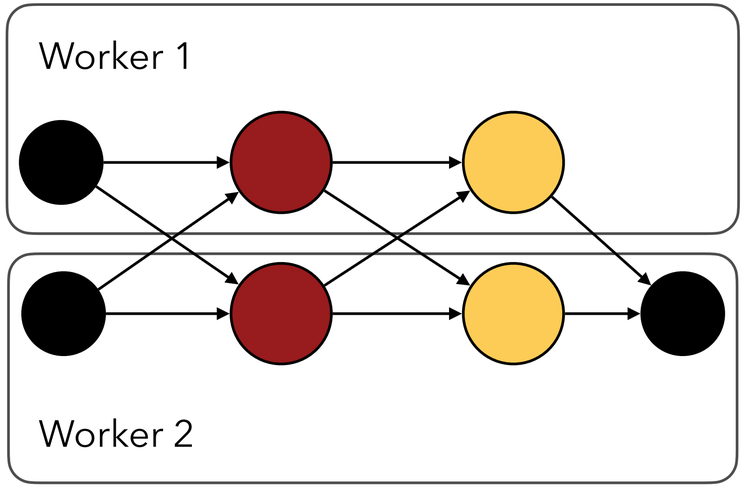
14 July 2020

STREAMING DATAFLOWS

Nexmark Q4: "Rolling average of winning bids"



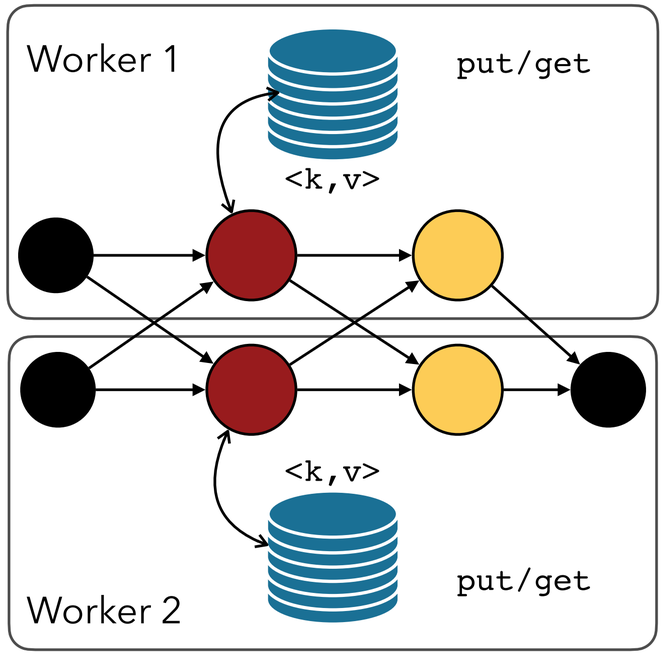
Logical Dataflow



Physical Dataflow

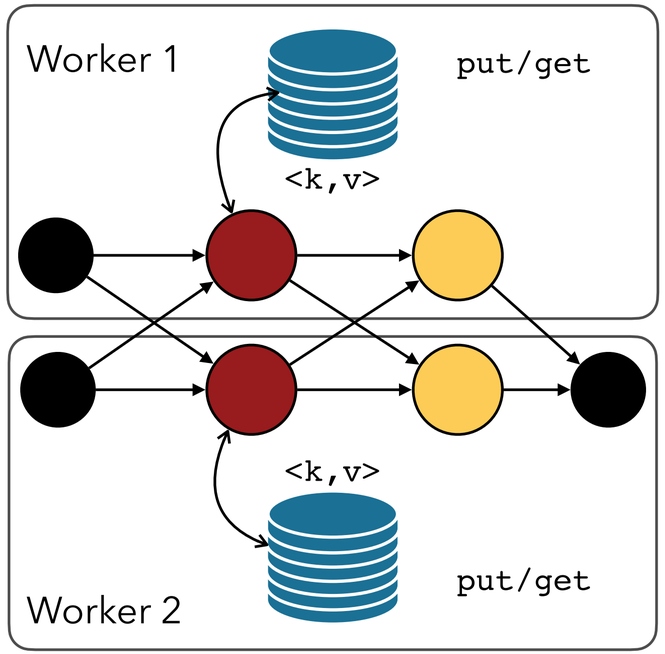
Nexmark Streaming Benchmark Suite: https://beam.apache.org/documentation/sdks/java/testing/nexmark/

LARGER-THAN-MEMORY STATE MANAGEMENT

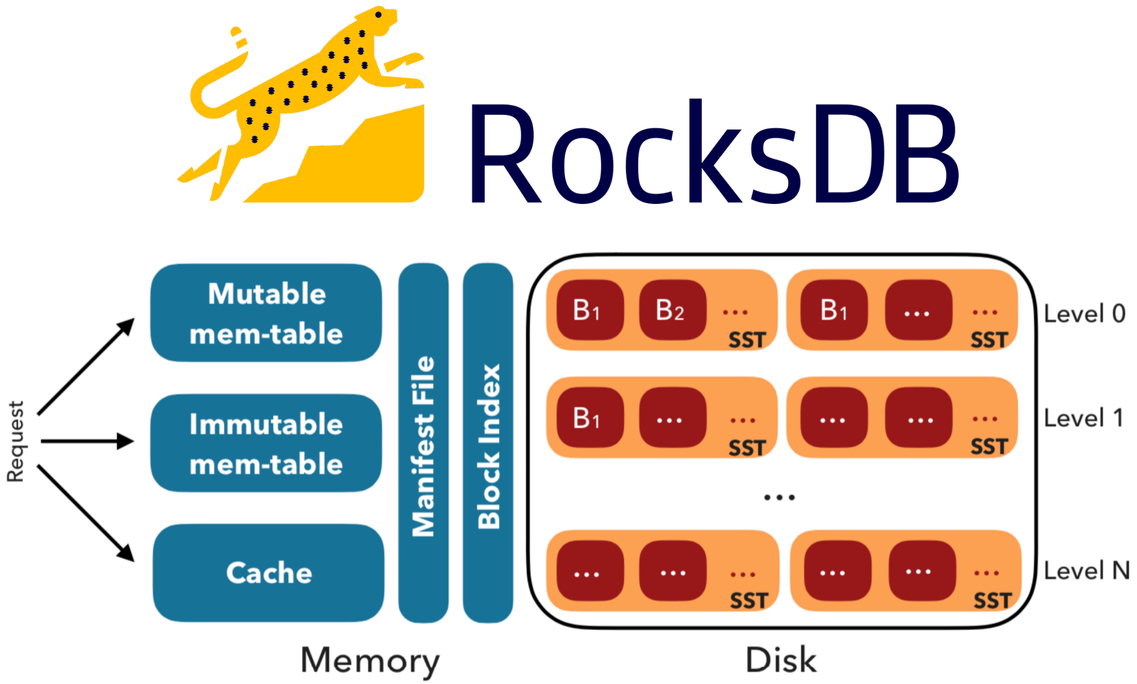


Large operator state is
backed by key-value stores

LARGER-THAN-MEMORY STATE MANAGEMENT



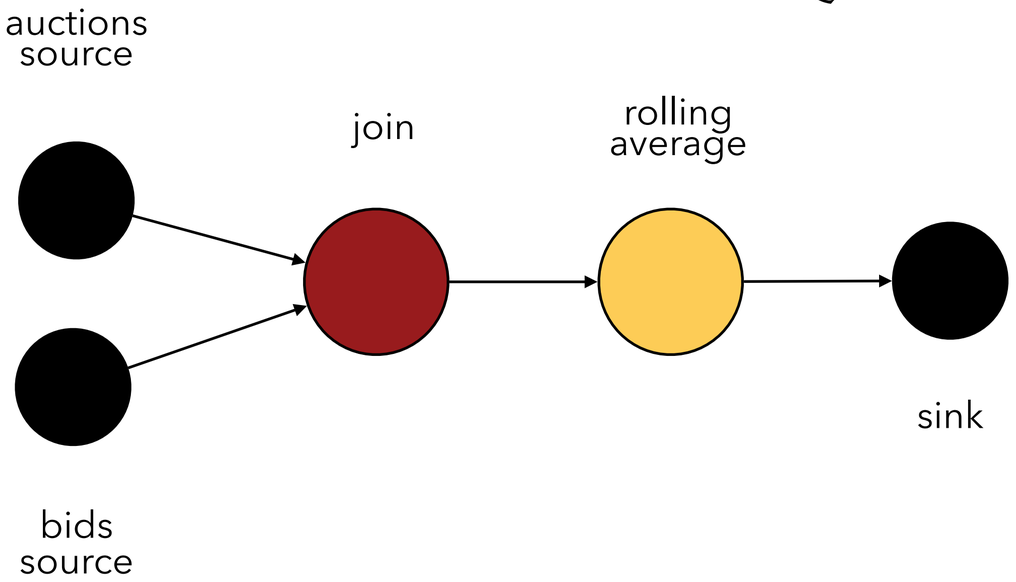
Large operator state is backed by key-value stores



LSM-based write-optimized store with efficient range scans

STATE REQUIREMENTS VARY ACROSS OPERATORS

Nexmark Q4: "Rolling average of winning bids"

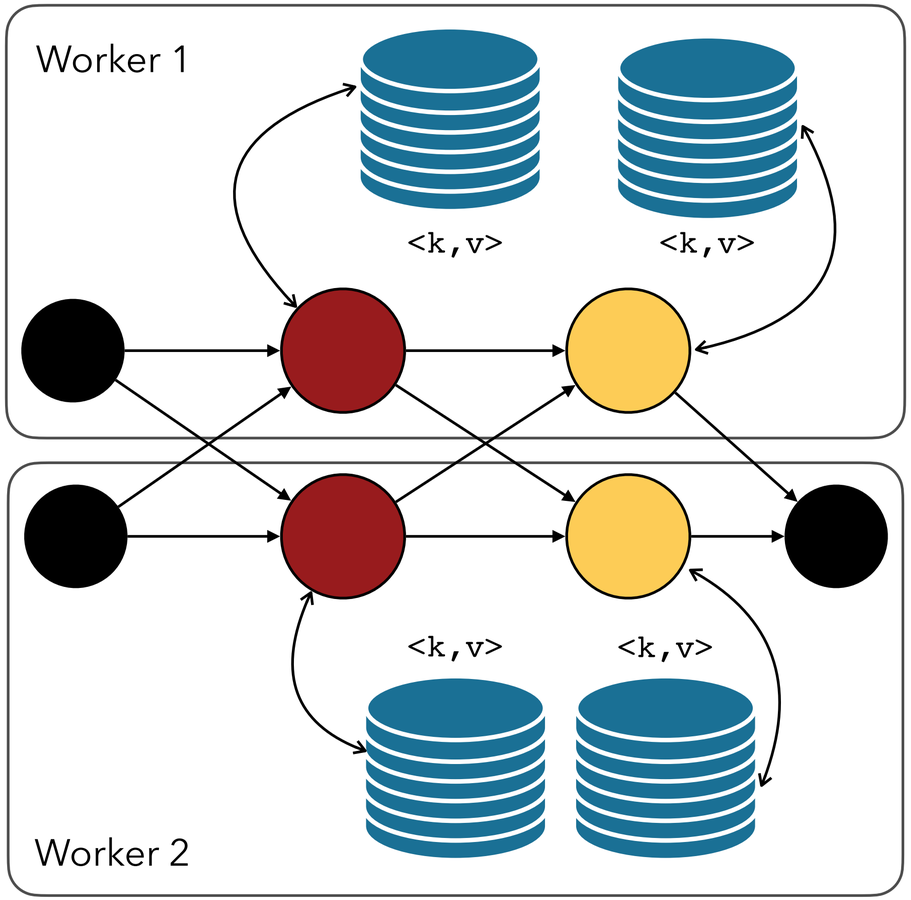


Join: Write-heavy and can potentially accumulate large state

Average: Read-Modify-Write a single value

Dataflow operators may have different *state access patterns* and *memory requirements*

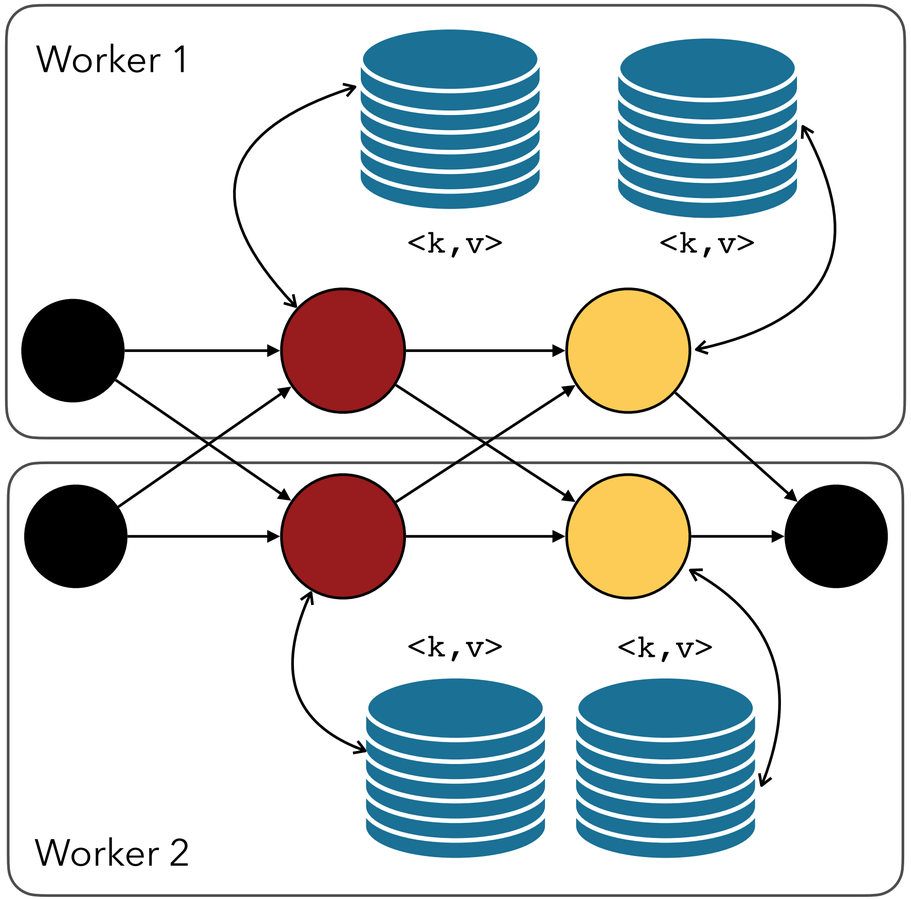
CURRENT PRACTICE: MONOLITHIC STATE MANAGEMENT



One key-value store (RocksDB) per stateful operator instance

All key-value stores in the dataflow are *globally-configured*

FLAWS OF MONOLITHIC STATE MANAGEMENT



- Oblivious store configuration
- Unnecessary data marshaling
- Unnecessary key-value store features

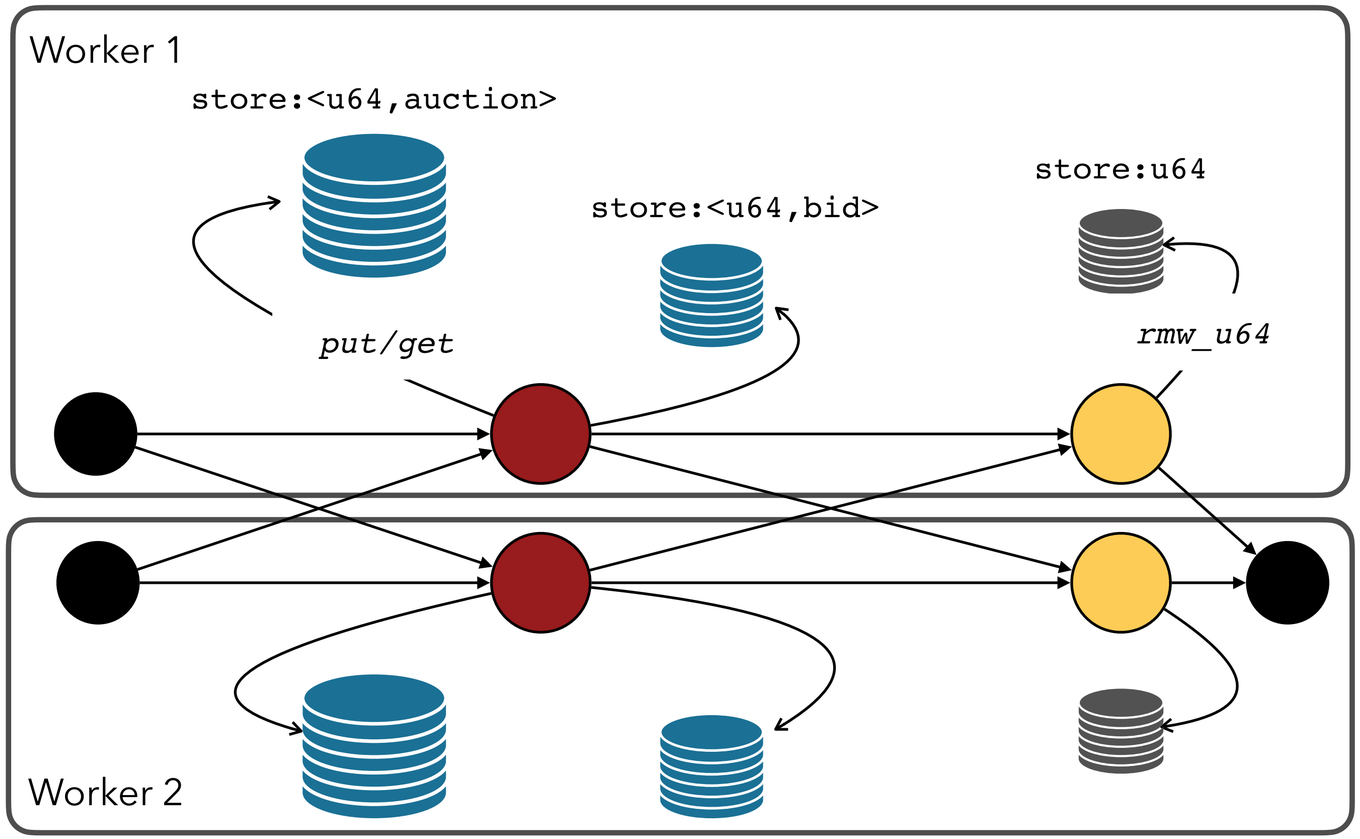
UNNECESSARY KEY-VALUE STORE FEATURES

- State partitioning
- State scoping
- Concurrent access to state
- State checkpointing

All these operations are handled by modern stream processors outside the state store

Stream processors guarantee single-thread access to state

WORKLOAD-AWARE STREAMING STATE MANAGEMENT



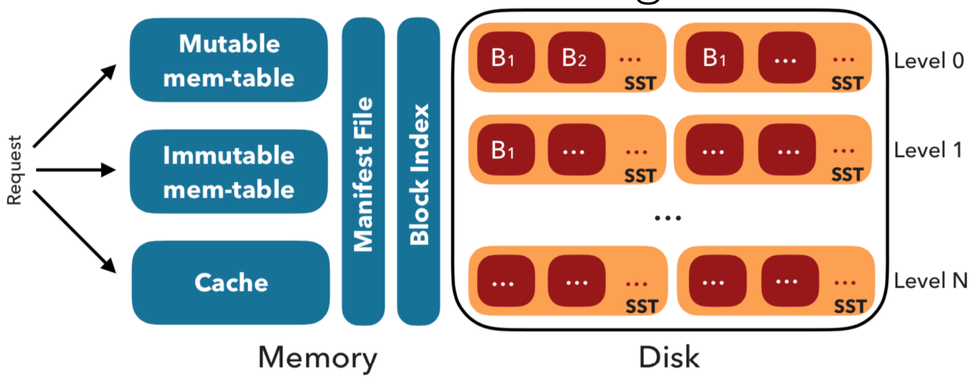
Multiple state stores of *different types* and *configurations* according to the requirements of the stateful operators

Streaming operators are *instantiated once* and are *long-running*: their access patterns and state sizes are largely known in advance

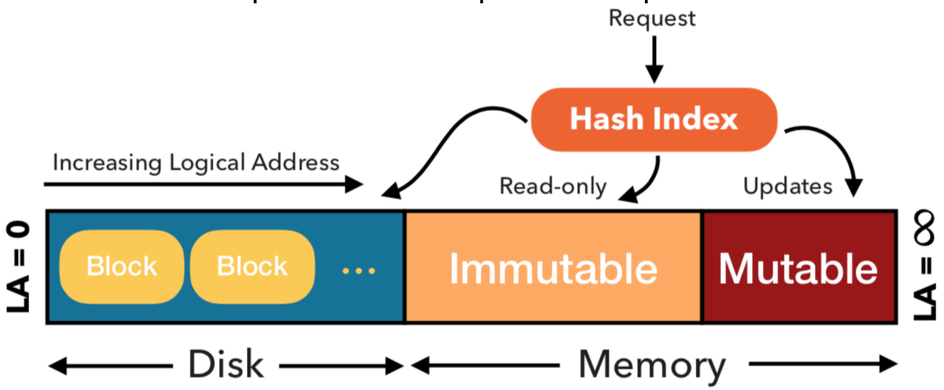
A FLEXIBLE TESTBED FOR STREAMING STATE MANAGEMENT

- Implemented in Rust
- Based on Timely Dataflow stream processor
- Supports two key-value stores
  - RocksDB
  - FASTER
- Supports different window evaluation strategies

RocksDB: LSM-based with efficient range scans



FASTER: Hybrid log with efficient lookups and in-place updates



Testbed: https://github.com/jliagouris/wassm

Timely Dataflow: https://github.com/TimelyDataflow/timely-dataflow

FASTER: https://github.com/microsoft/FASTER

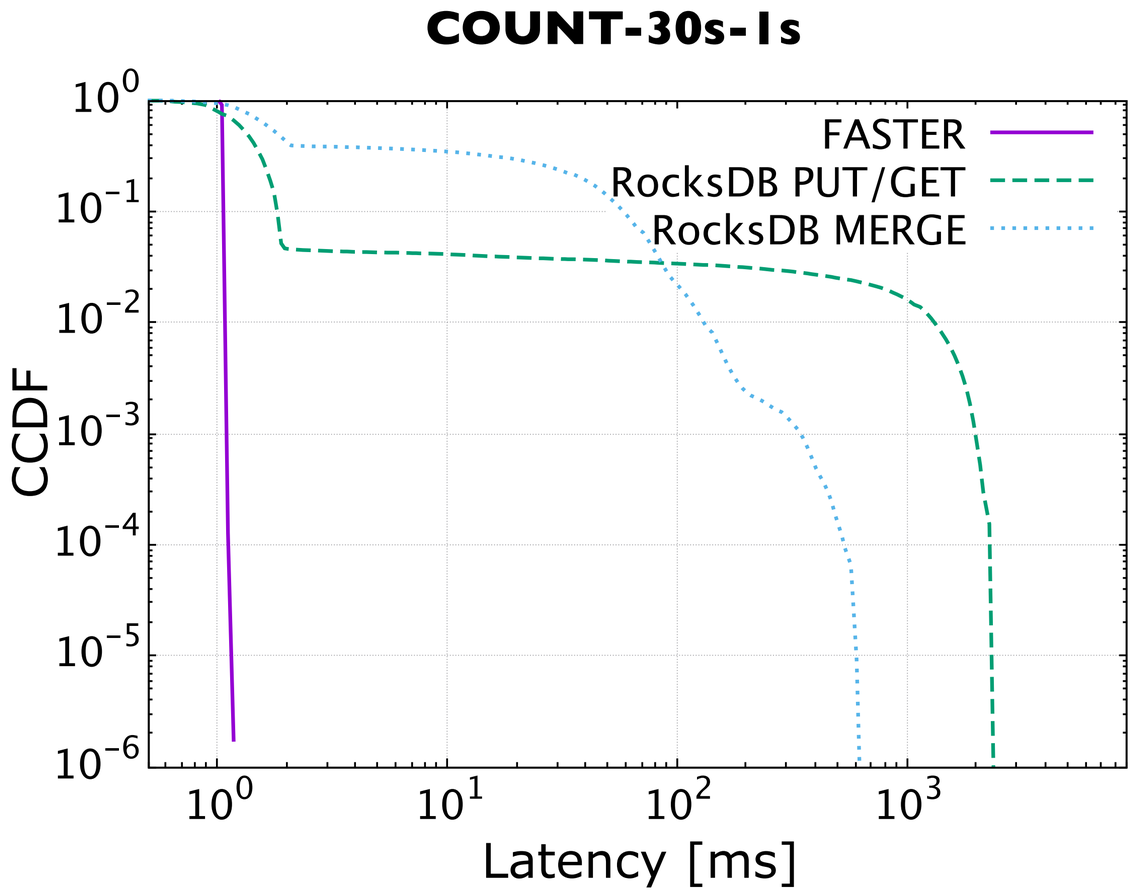
EXPERIMENTAL RESULTS

EVALUATION GOALS

1. Study the effect of the backend's data layout on the evaluation of streaming windows

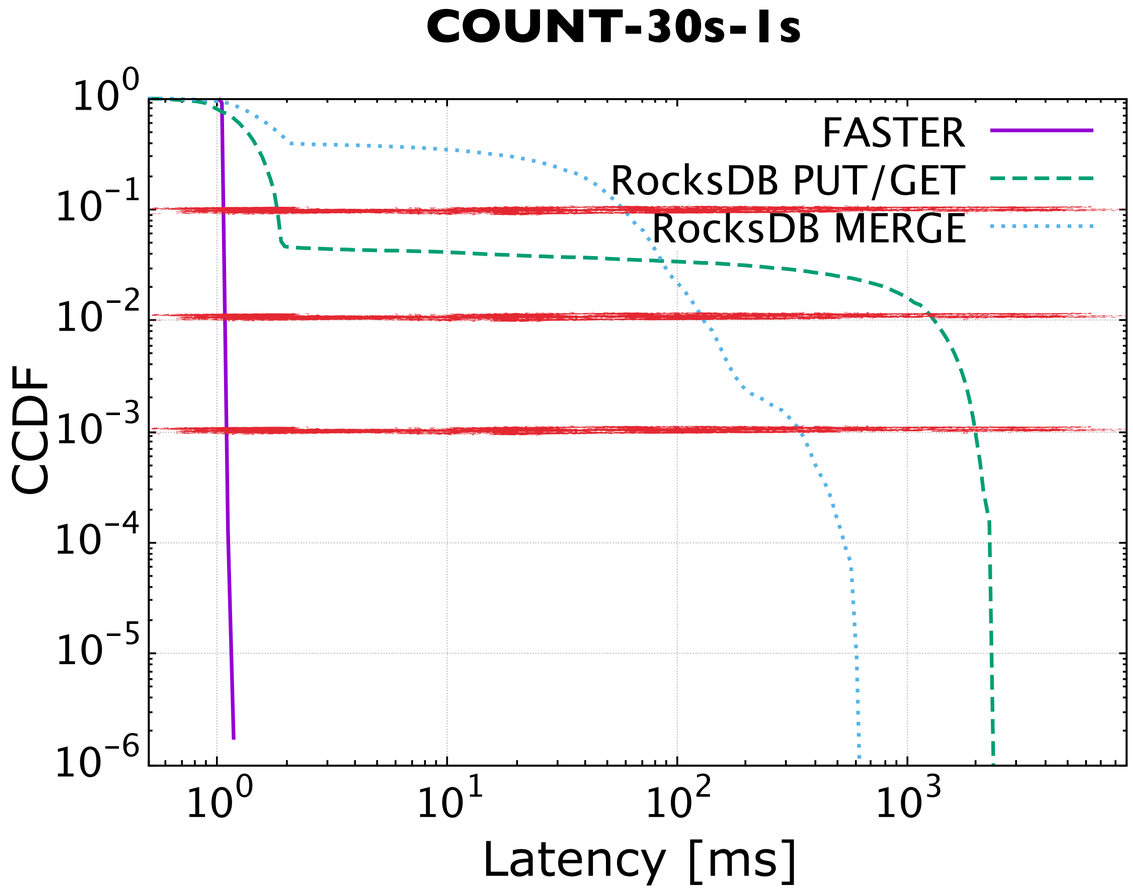
2. Study the effect of workload-aware configuration on queries with multiple stateful operators

EFFECT OF DATA LAYOUT ON WINDOW EVALUATION



- Query 1: Count the number of records in a 30s window that slides every 1s
- Input rate: 10K records/s
- Single thread execution
- Report end-to-end latency (ms) per record

EFFECT OF DATA LAYOUT ON WINDOW EVALUATION



p90

p99

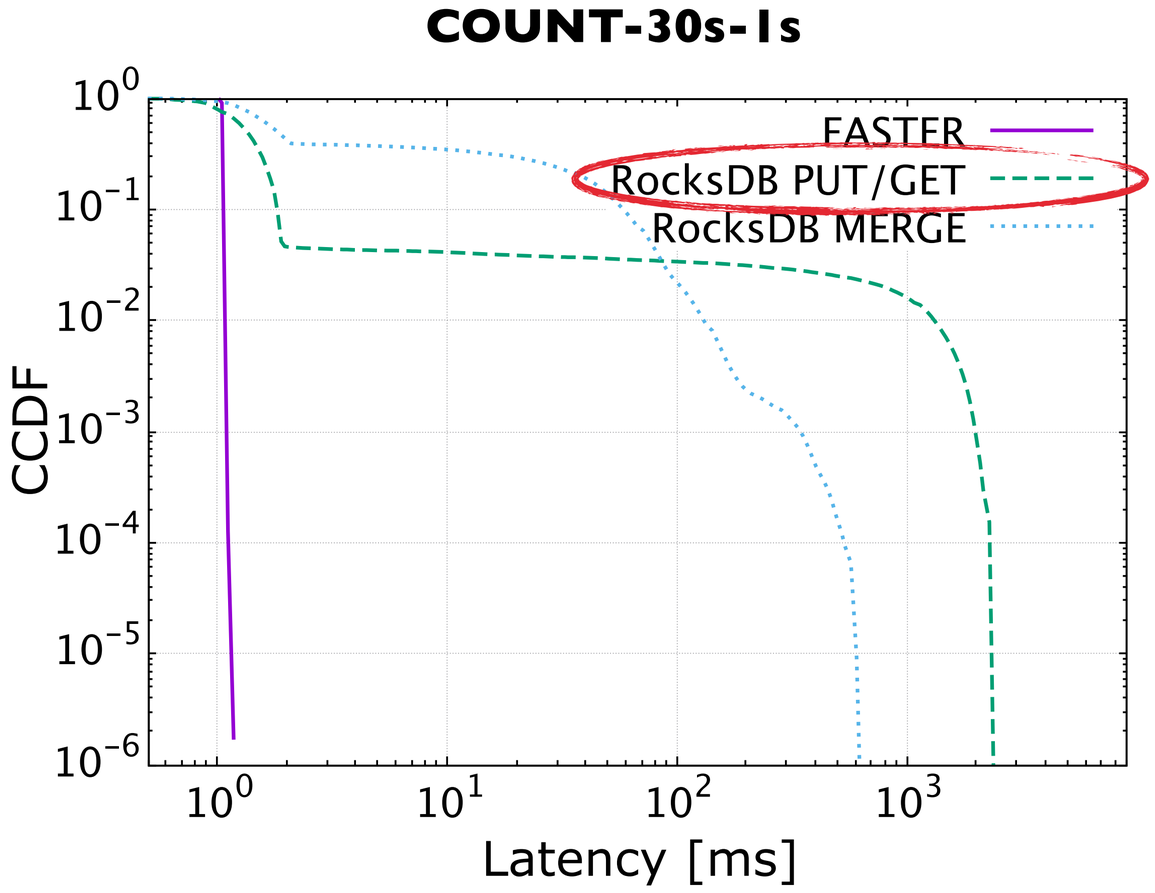
p99.9

…

Lower is better

Complementary CDF: Each point *(x,y)* indicates that *y*% of the latency measurements are at least *x ms*

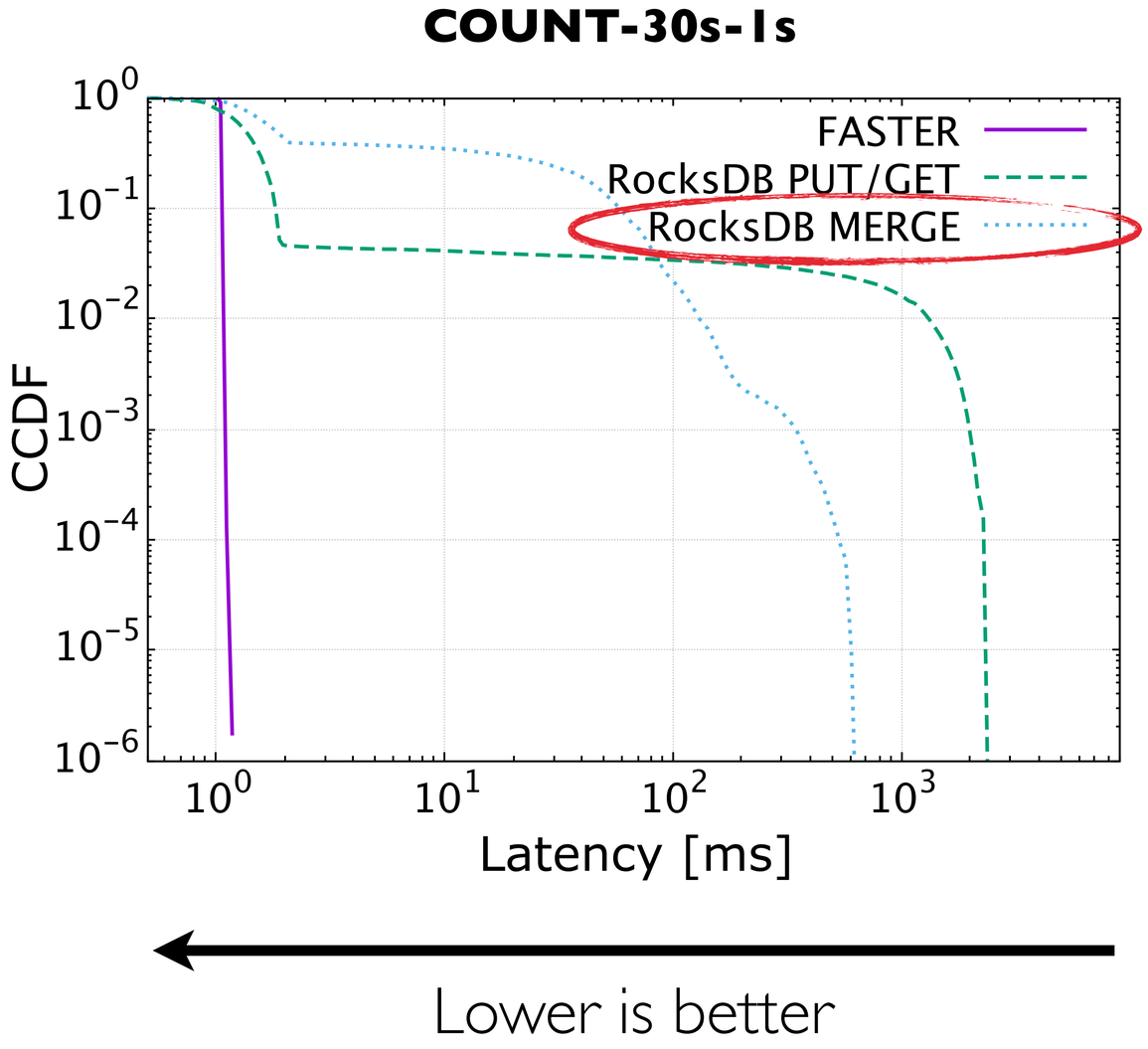
EFFECT OF DATA LAYOUT ON WINDOW EVALUATION



Lower is better

RocksDB PUT/GET: *On record*, retrieve window contents, apply new record, and put the updated contents back to the store

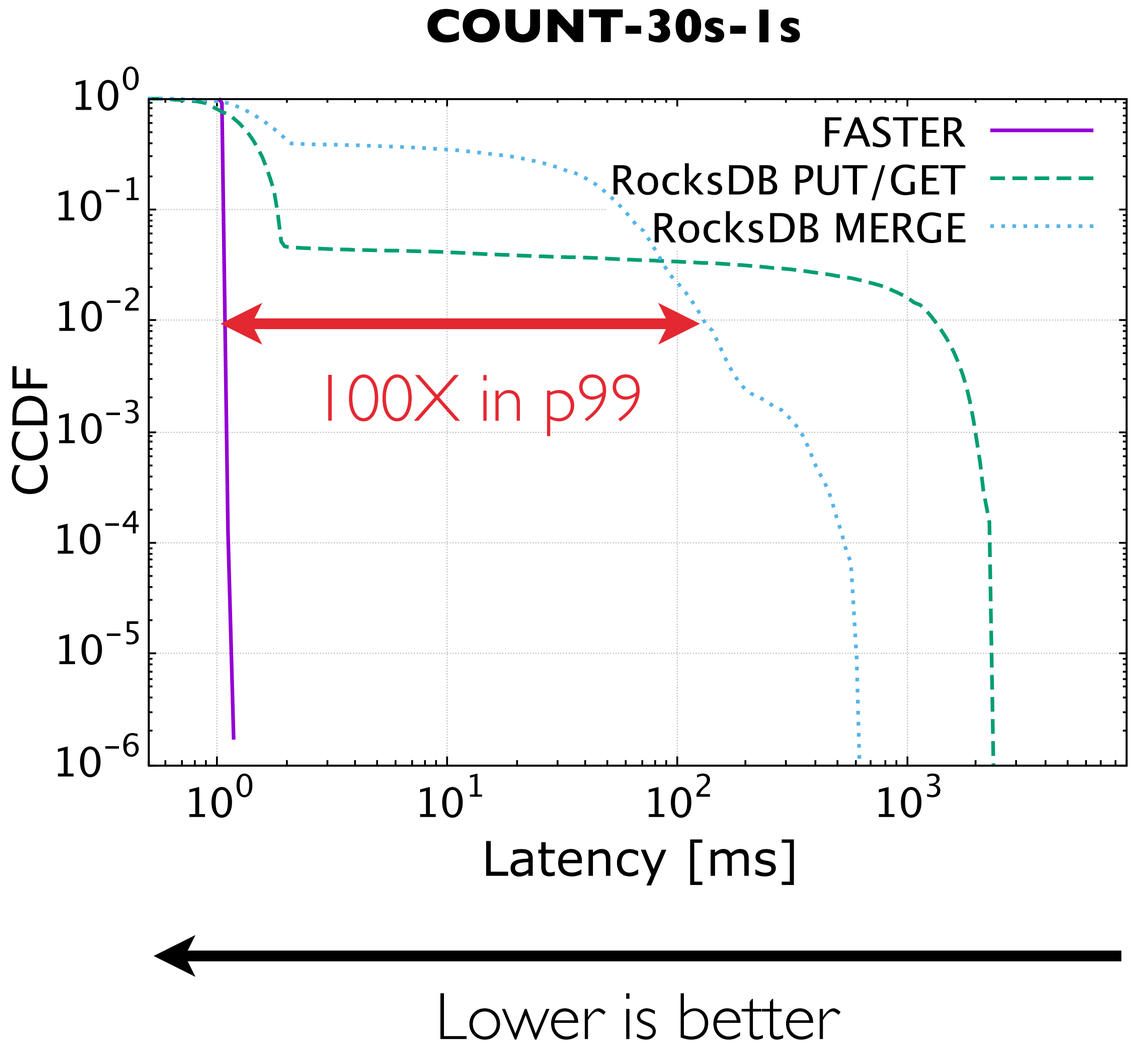
EFFECT OF DATA LAYOUT ON WINDOW EVALUATION



RocksDB MERGE: *On record*, put record to the store using MERGE.

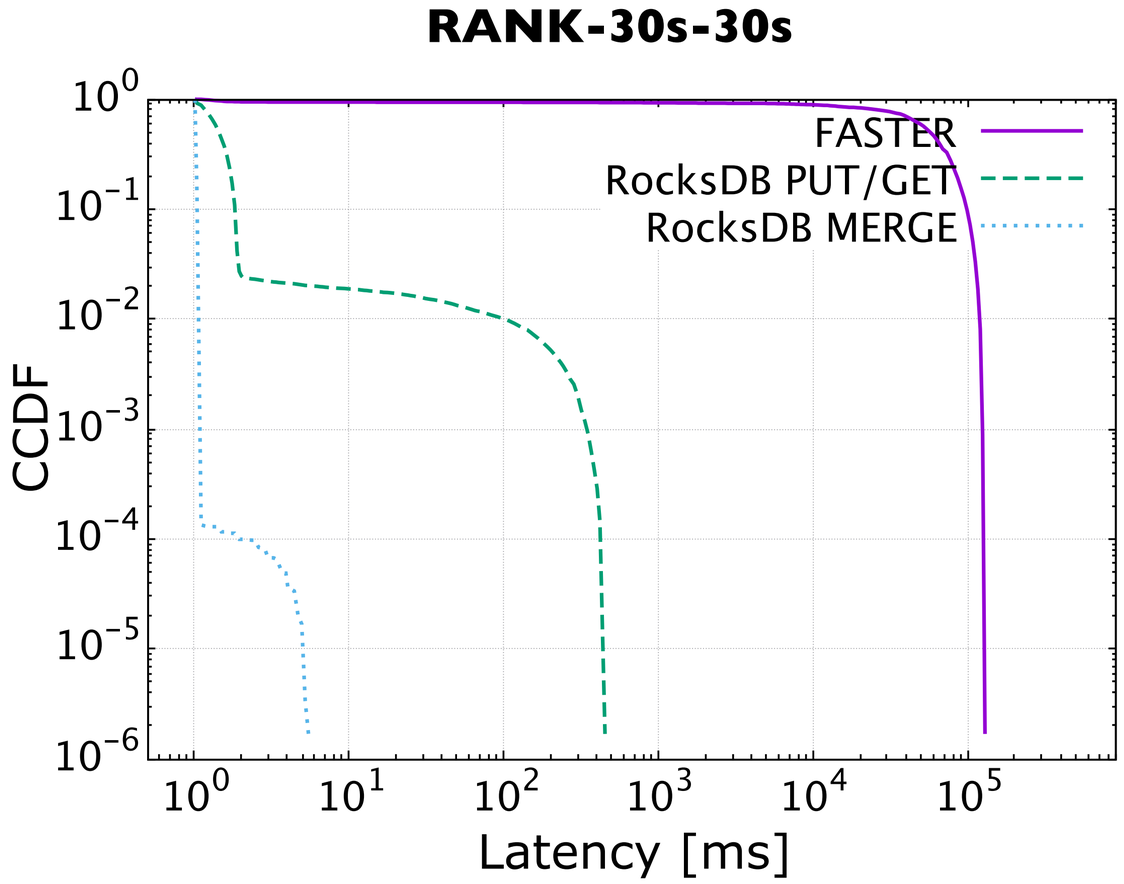
The record is applied to the window contents *lazily on trigger*

EFFECT OF DATA LAYOUT ON WINDOW EVALUATION



FASTER performs better due to ***in-place updates***

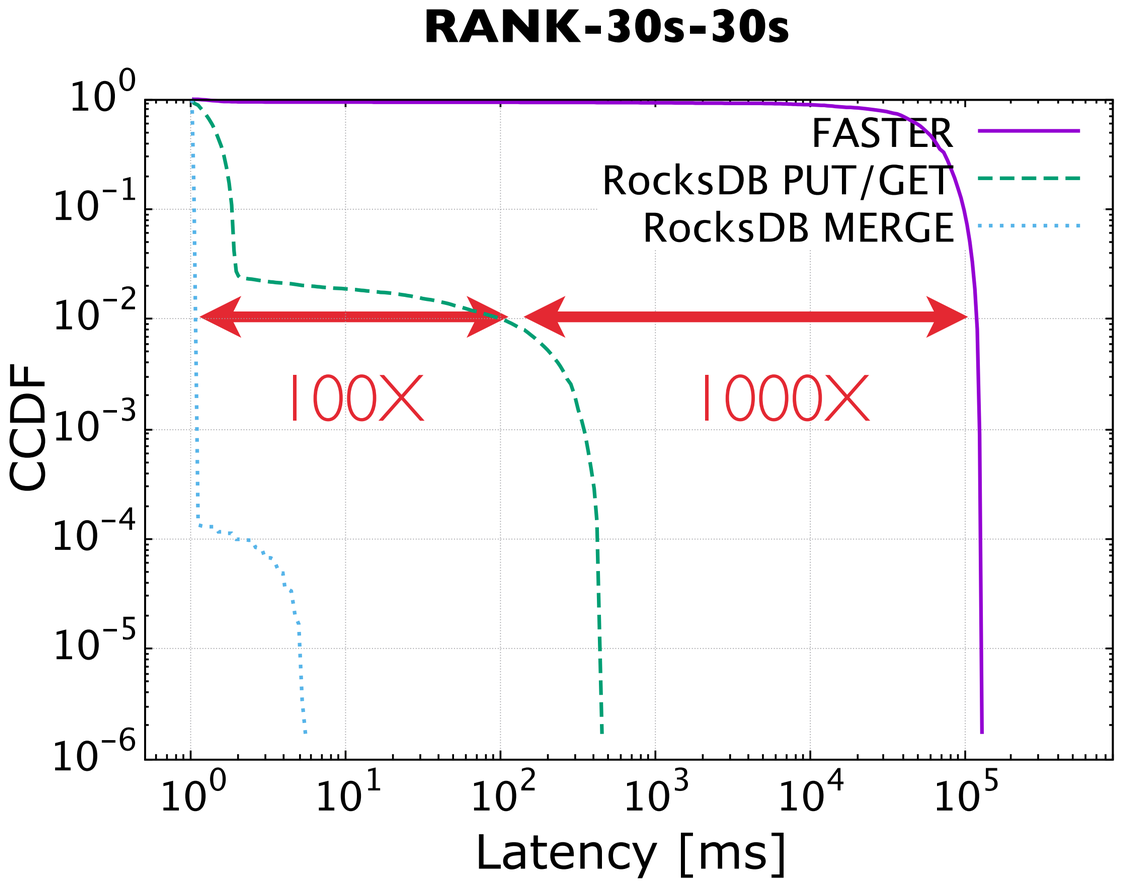
EFFECT OF DATA LAYOUT ON WINDOW EVALUATION



Lower is better

- Query 2: Rank records in a 30s tumbling window
- Input rate: 1K records/s
- Single thread execution
- Report end-to-end latency (ms) per record

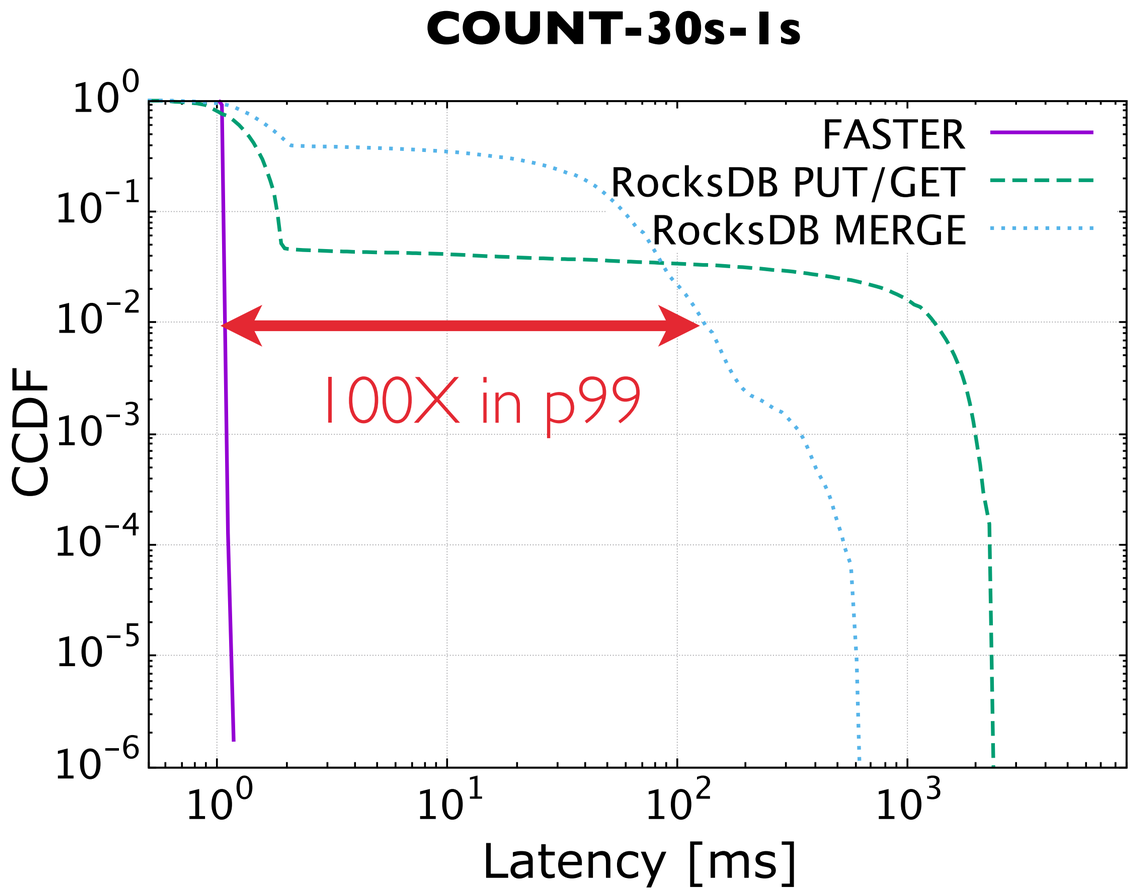
EFFECT OF DATA LAYOUT ON WINDOW EVALUATION

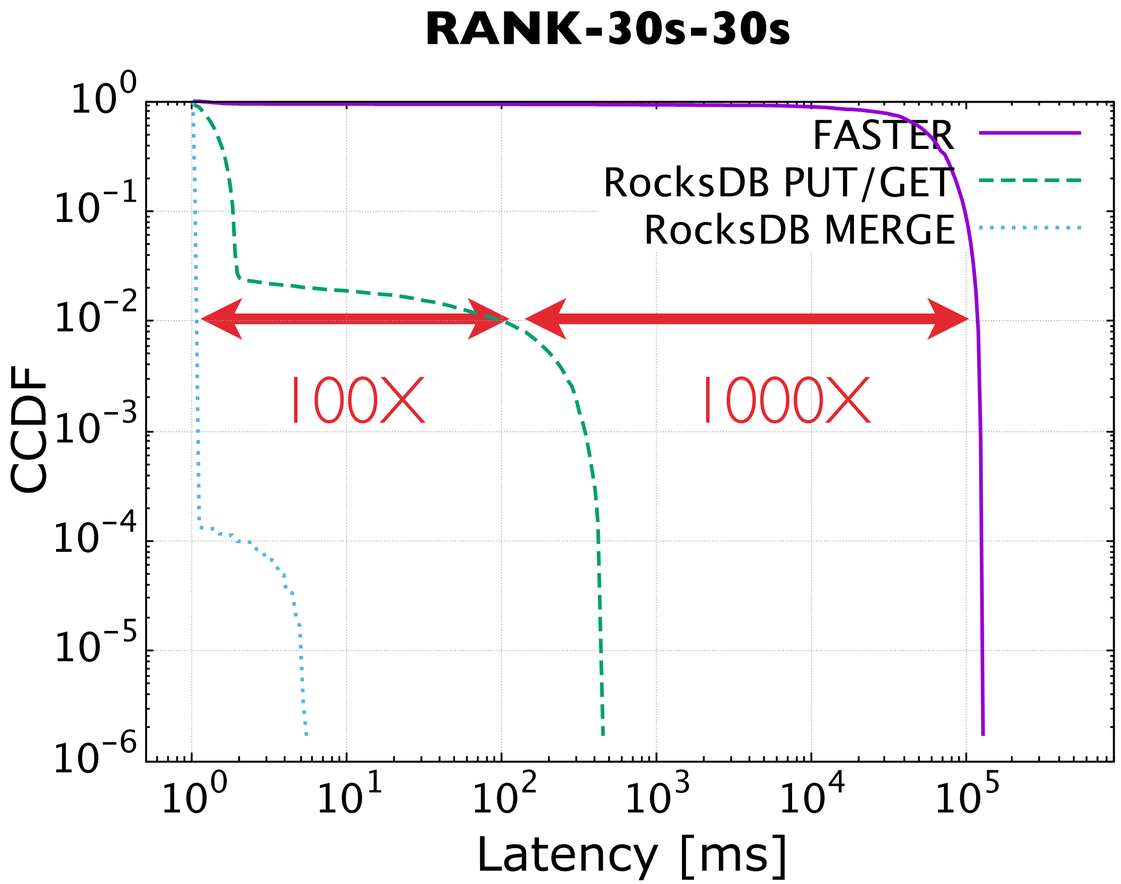


Lower is better

RocksDB MERGE performs best due to ***lazy evaluation***

THERE IS NO CLEAR WINNER



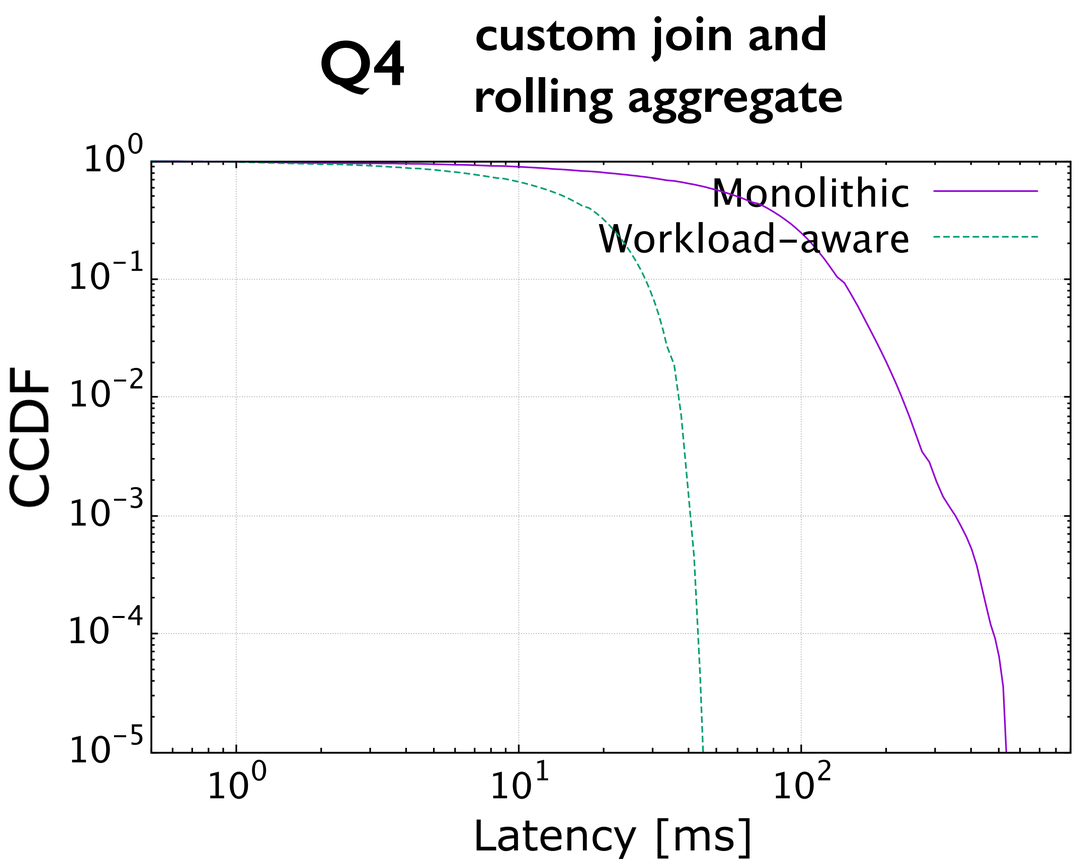


MONOLITHIC VS WORKLOAD-WARE STATE MANAGEMENT

- Experiments with six Nexmark* queries
- Different stateful operators (*joins*, *window aggregations*, *custom aggregations*)
- Simple workload-aware configuration of *data types* and *available memory size*

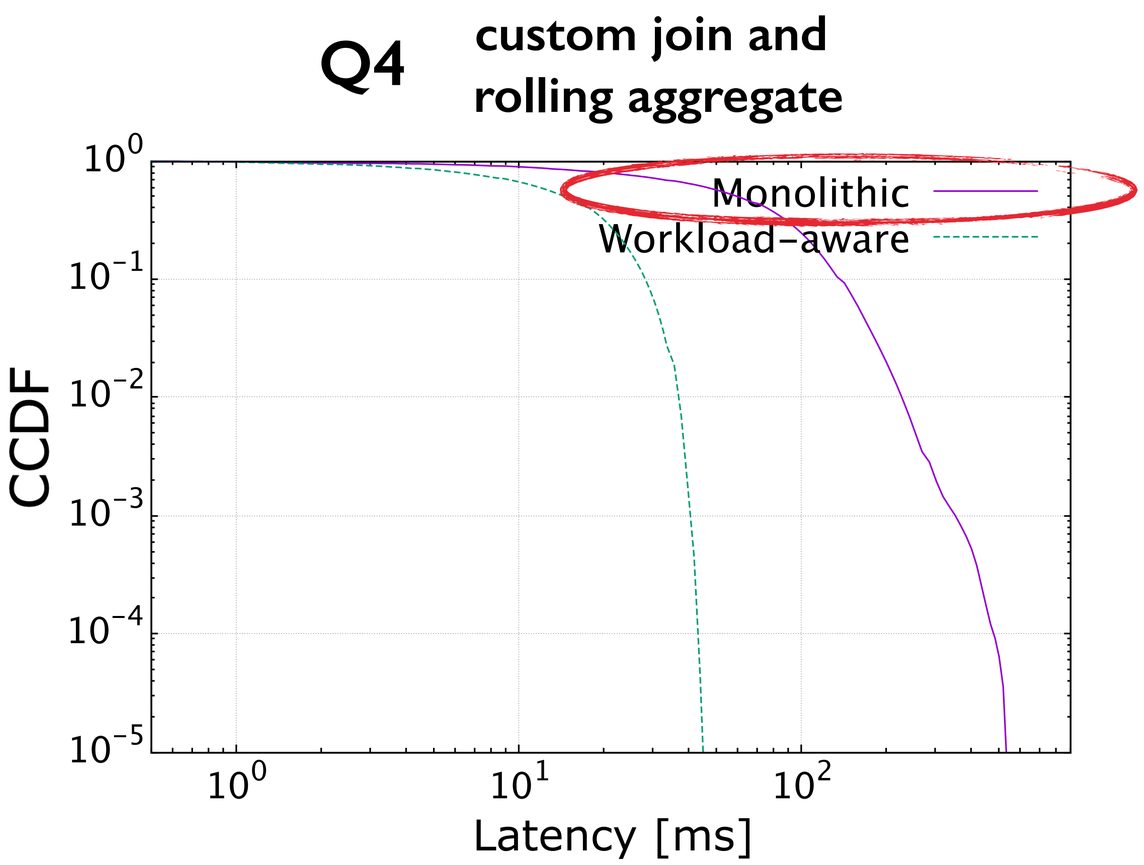
*Nexmark Streaming Benchmark Suite: https://beam.apache.org/documentation/sdks/java/testing/nexmark/

MONOLITHIC VS WORKLOAD-AWARE STATE MANAGEMENT



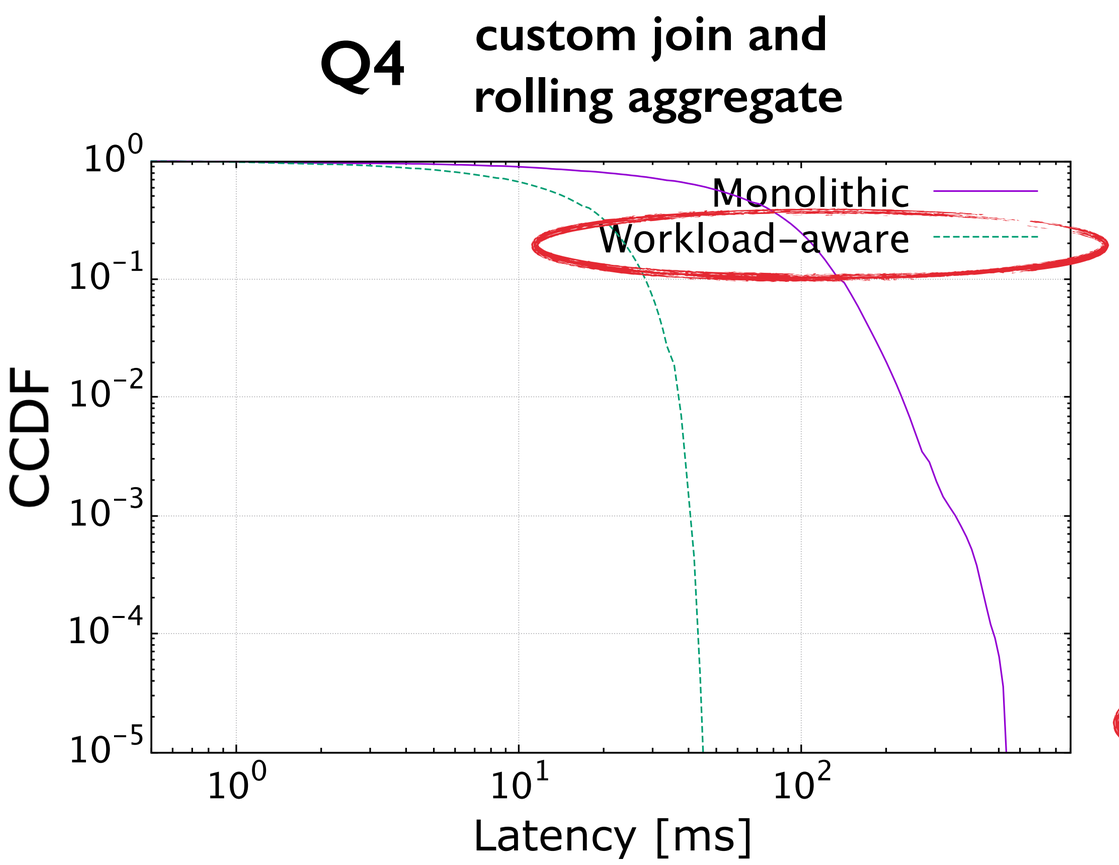
- State store used: FASTER
- Input rate: 10K records/s
- SIngle thread execution
- Monolithic memory configuration: 8GB
- Workload-aware memory configuration: 6GB (bids), 1.5GB (auctions), 512MB (average)
- Report end-to-end latency (ms) per record

MONOLITHIC VS WORKLOAD-AWARE STATE MANAGEMENT



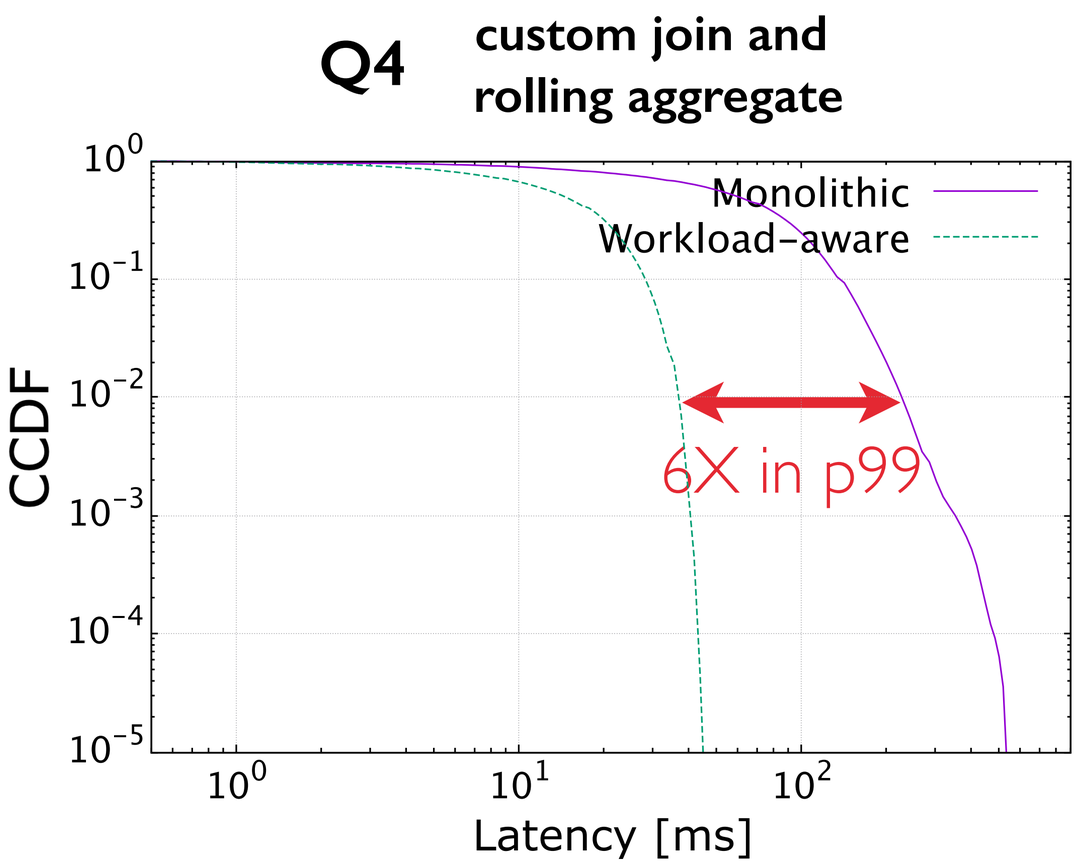
- State store used: FASTER
- Input rate: 10K records/s
- SIngle thread execution
- Monolithic memory configuration: 8GB
- Workload-aware memory configuration: 6GB (bids), 1.5GB (auctions), 512MB (average)
- Report end-to-end latency (ms) per record

MONOLITHIC VS WORKLOAD-AWARE STATE MANAGEMENT



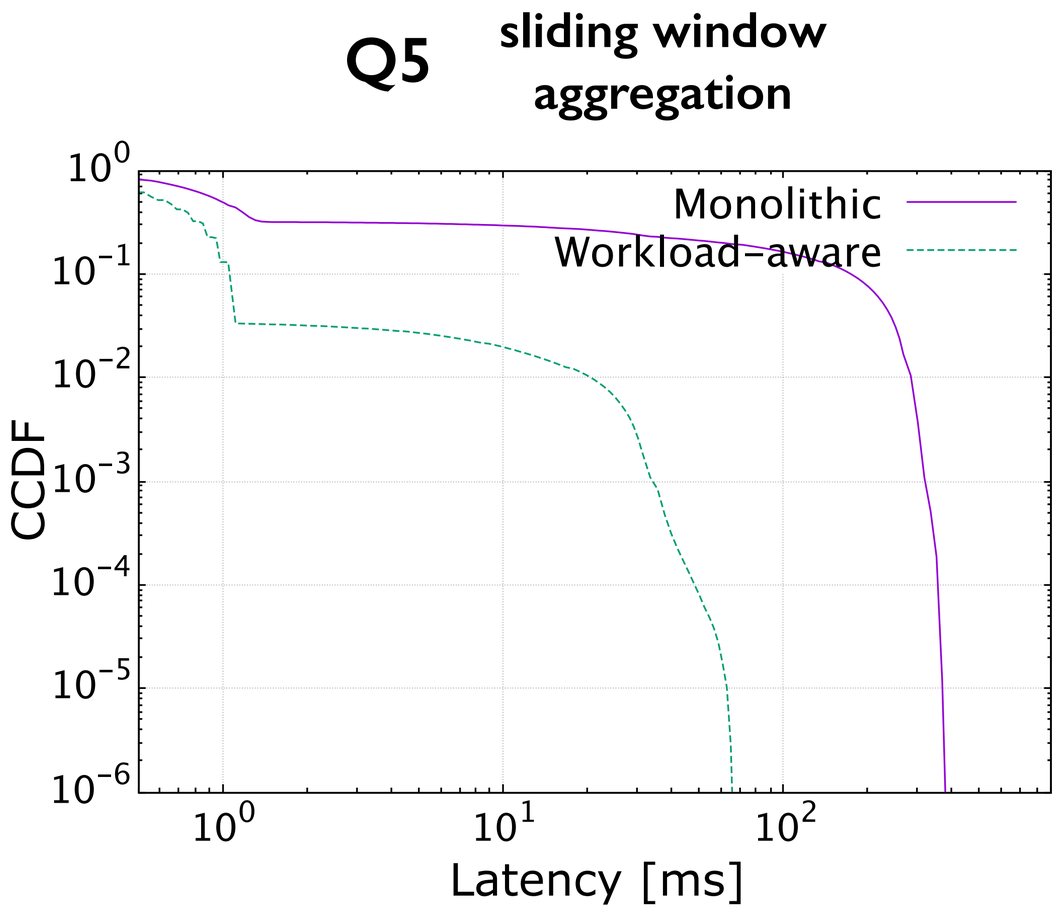
- State store used: FASTER
- Input rate: 10K records/s
- SIngle thread execution
- Monolithic memory configuration: 8GB
- Workload-aware memory configuration: 6GB (bids), 1.5GB (auctions), 512MB (average)
- Report end-to-end latency (ms) per record

MONOLITHIC VS WORKLOAD-AWARE STATE MANAGEMENT



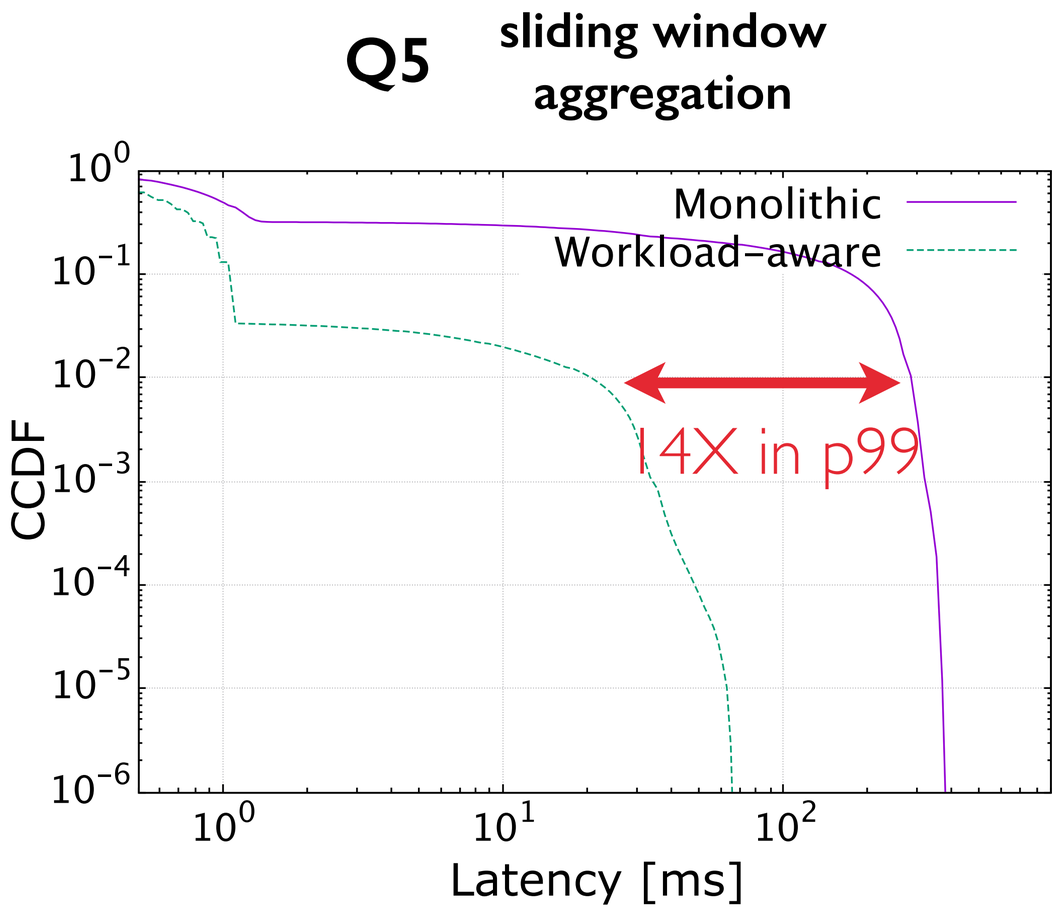
- State store used: FASTER
- Input rate: 10K records/s
- SIngle thread execution
- Monolithic memory configuration: 8GB
- Workload-aware memory configuration: 6GB (bids), 1.5GB (auctions), 512MB (average)
- Report end-to-end latency (ms) per record

MONOLITHIC VS WORKLOAD-AWARE STATE MANAGEMENT



- State store used: FASTER
- Input rate: 10K records/s
- Single thread execution
- Monolithic memory configuration: 8GB
- Workload-aware memory configuration: 6GB (additions), 1GB (deletions), 512MB (accumulations), 512MB (hot items)
- Report end-to-end latency (ms) per record

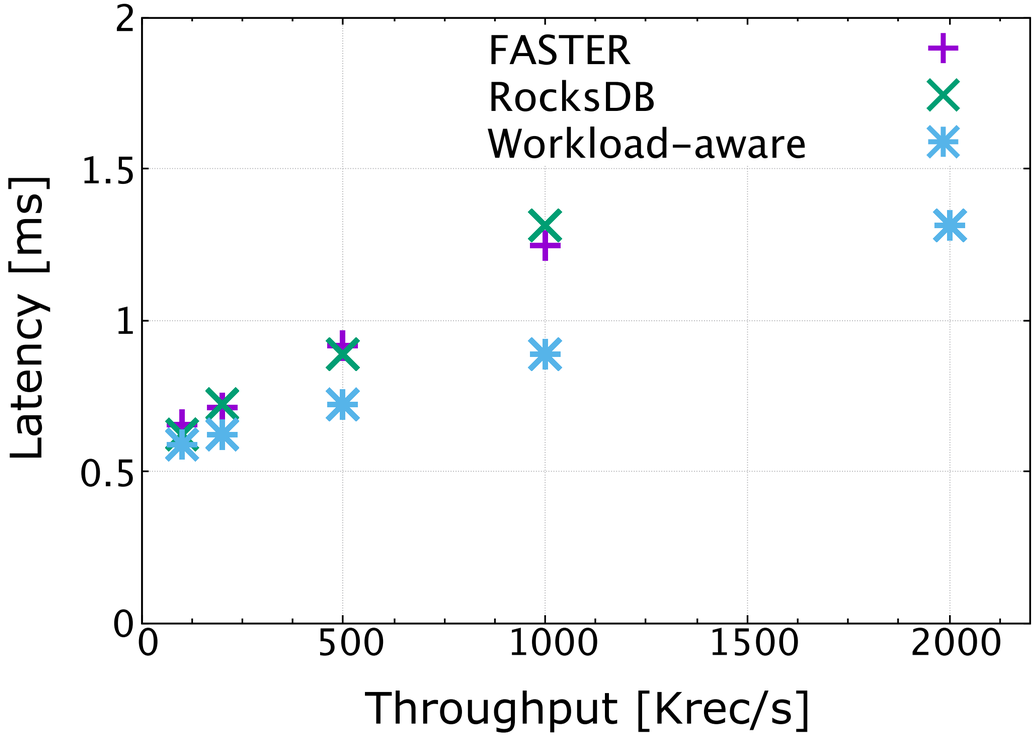
MONOLITHIC VS WORKLOAD-AWARE STATE MANAGEMENT



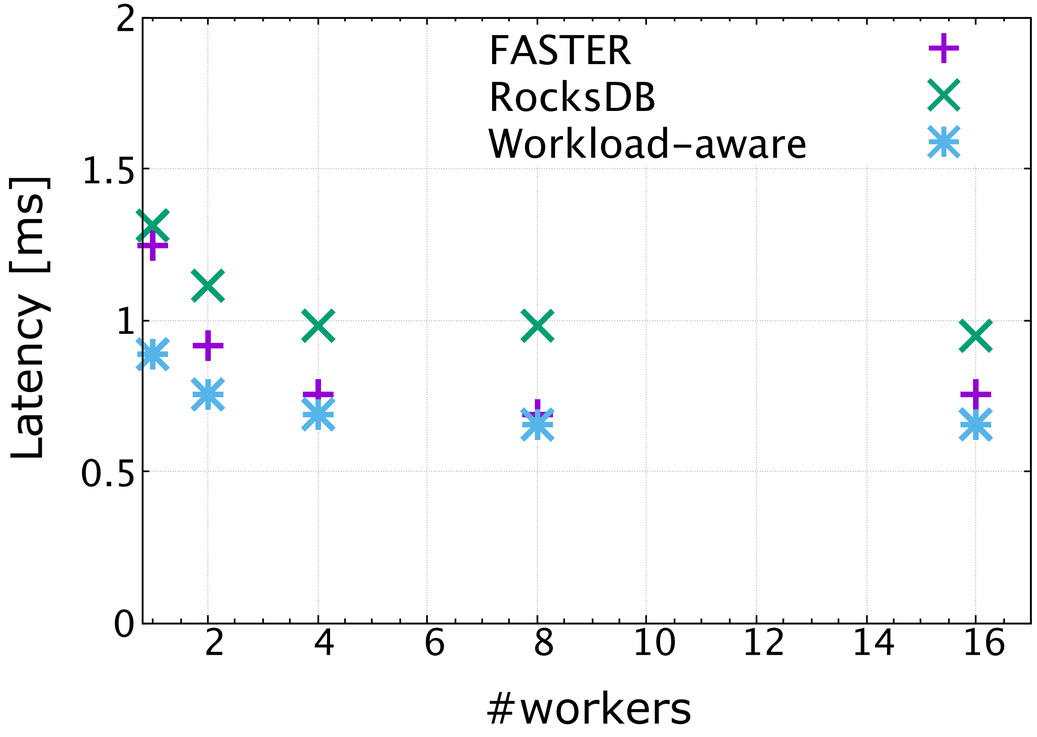
- State store used: FASTER
- Input rate: 10K records/s
- Single thread execution
- Monolithic memory configuration: 8GB
- Workload-aware memory configuration: 6GB (additions), 1GB (deletions), 512MB (accumulations), 512MB (hot items)
- Report end-to-end latency (ms) per record

MONOLITHIC VS WORKLOAD-AWARE STATE MANAGEMENT

Q4 latency vs throughput with a single thread

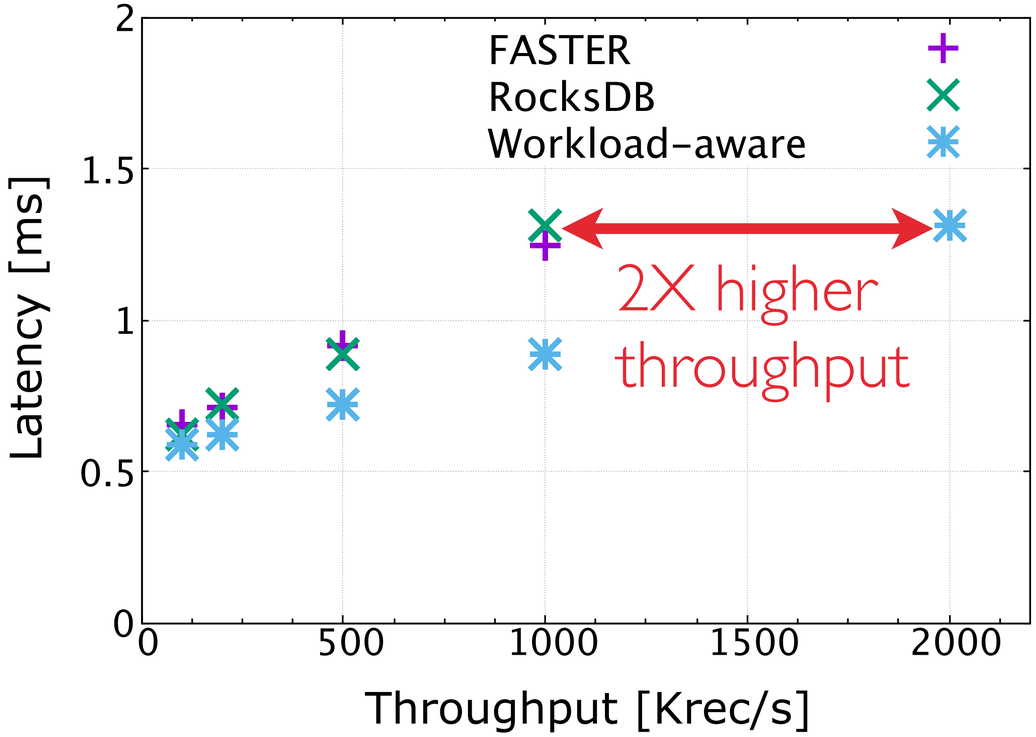


Q7 latency with varying the number of threads

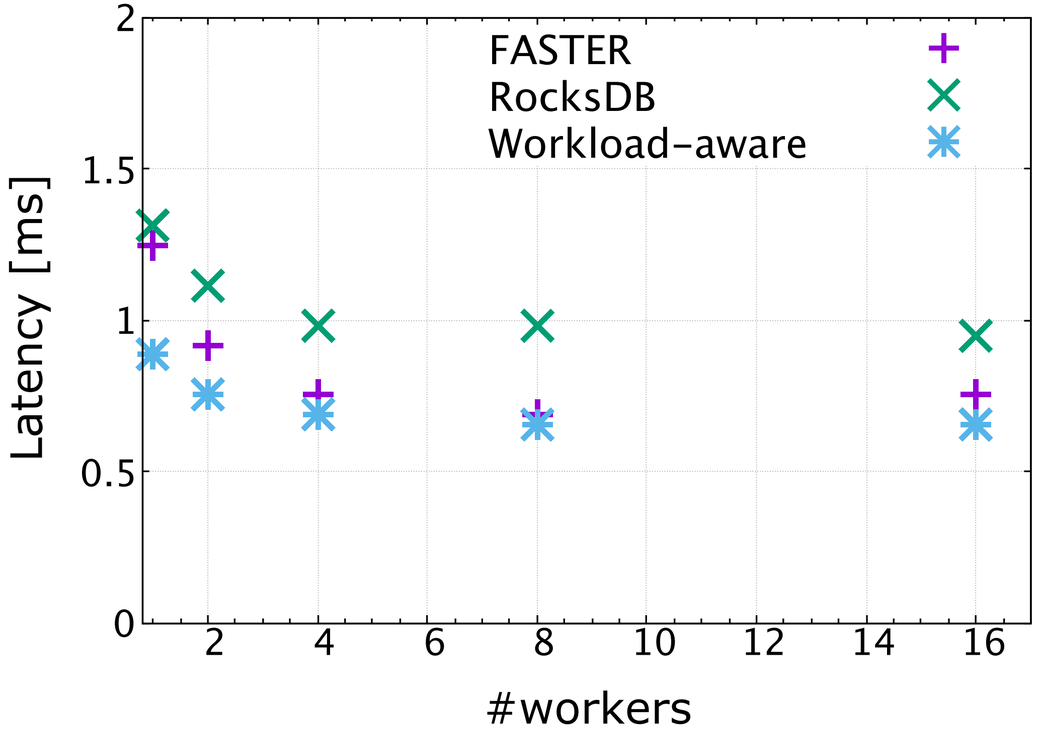


MONOLITHIC VS WORKLOAD-AWARE STATE MANAGEMENT

Q4 latency vs throughput with a single thread

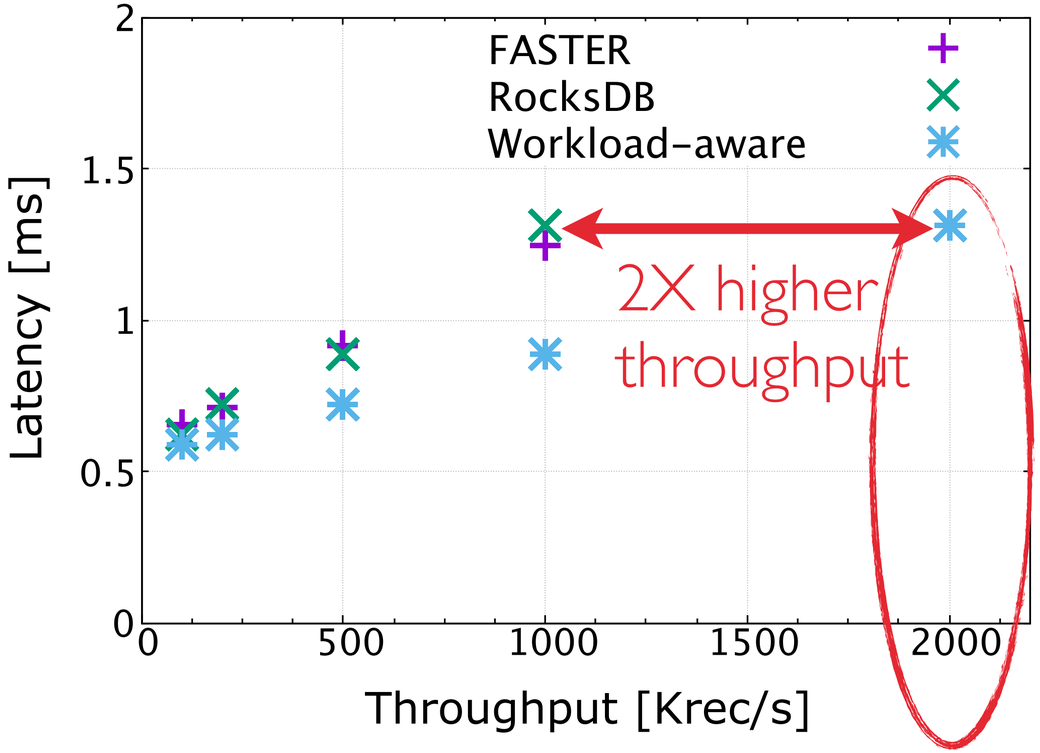


Q7 latency with varying the number of threads

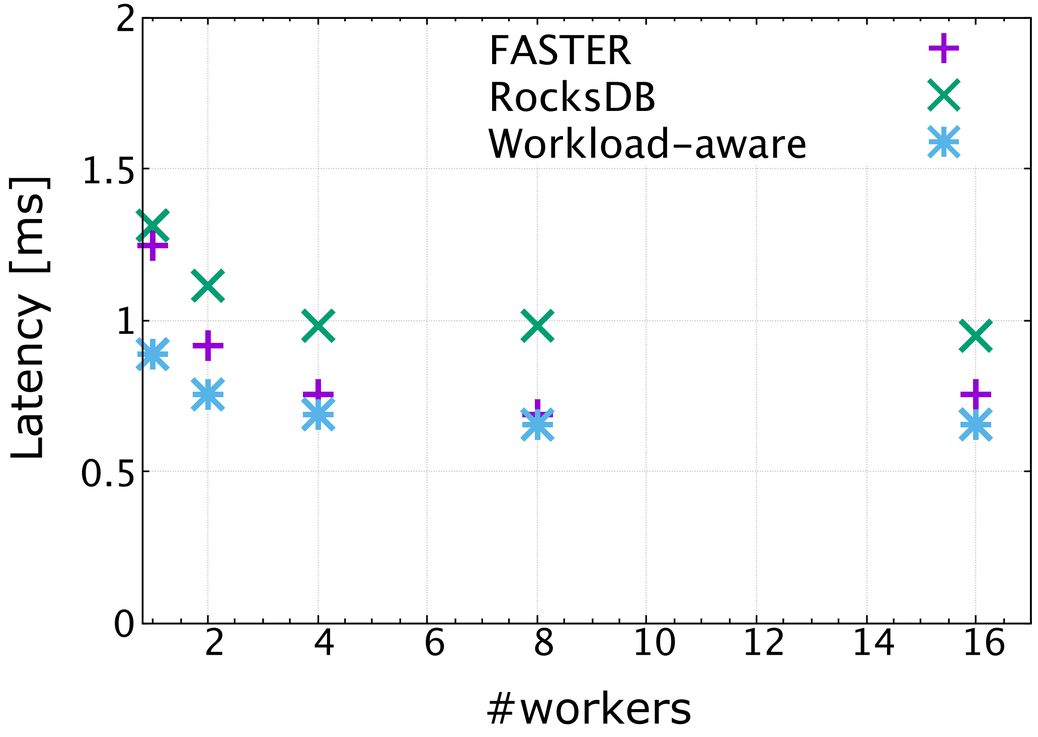


MONOLITHIC VS WORKLOAD-AWARE STATE MANAGEMENT

Q4 latency vs throughput with a single thread



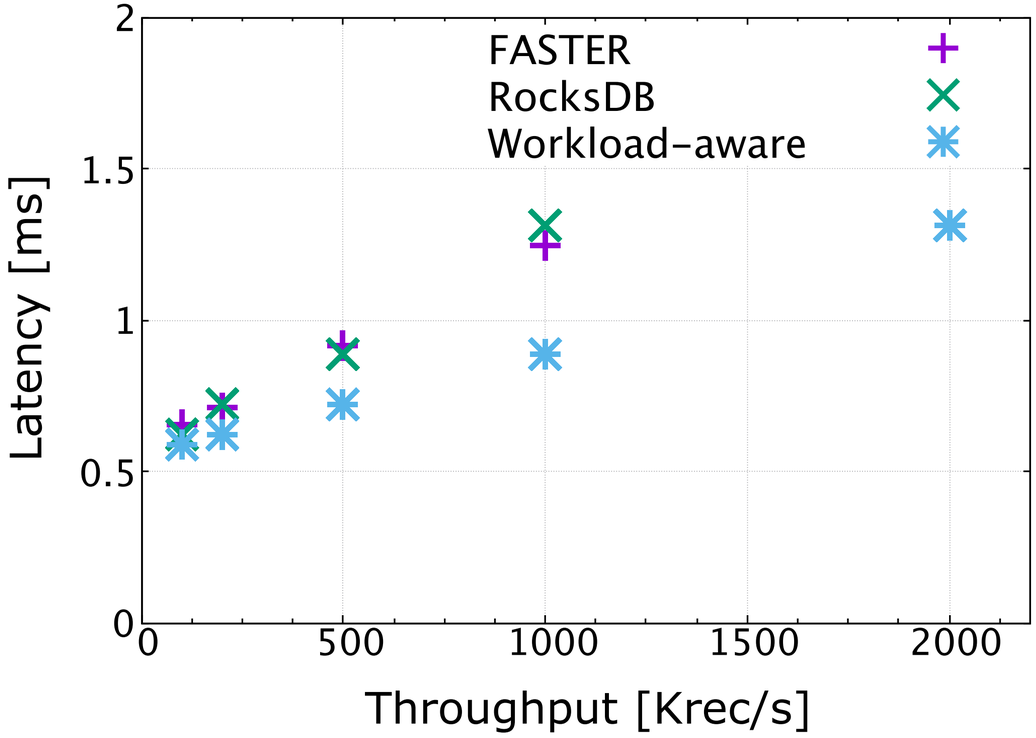
Q7 latency with varying the number of threads



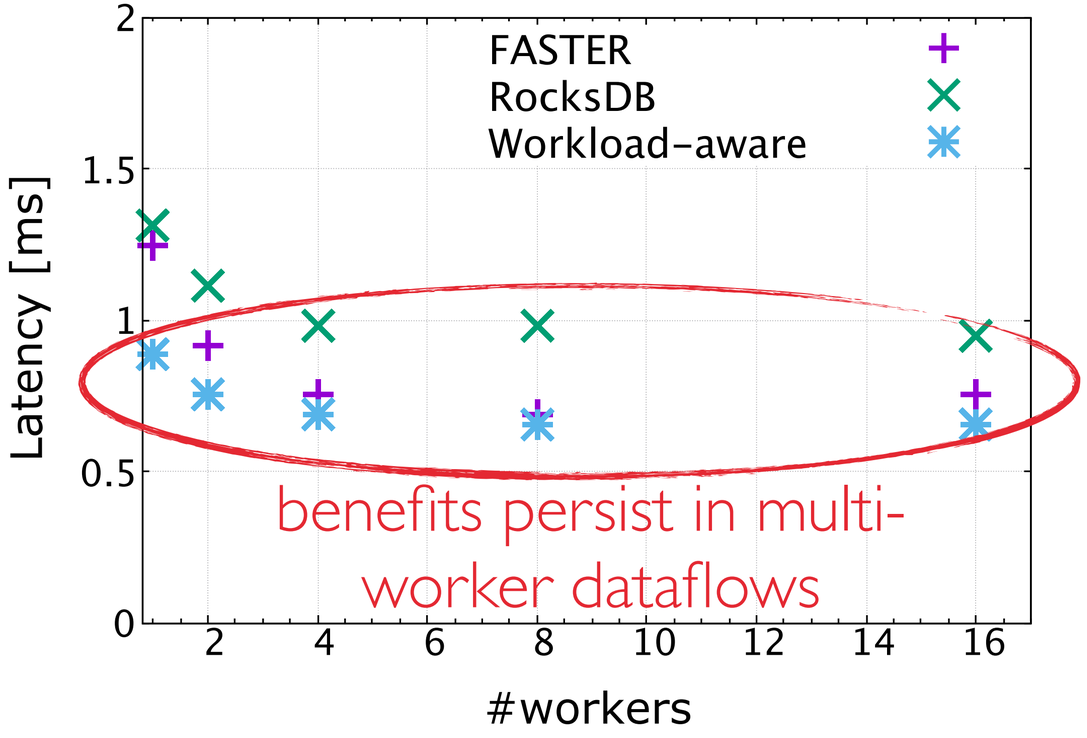
FASTER (monolithic) and RocksDB (monolithic) do not keep up with 2M records/s

MONOLITHIC VS WORKLOAD-AWARE STATE MANAGEMENT

Q4 latency vs throughput with a single thread



Q7 latency with varying the number of threads



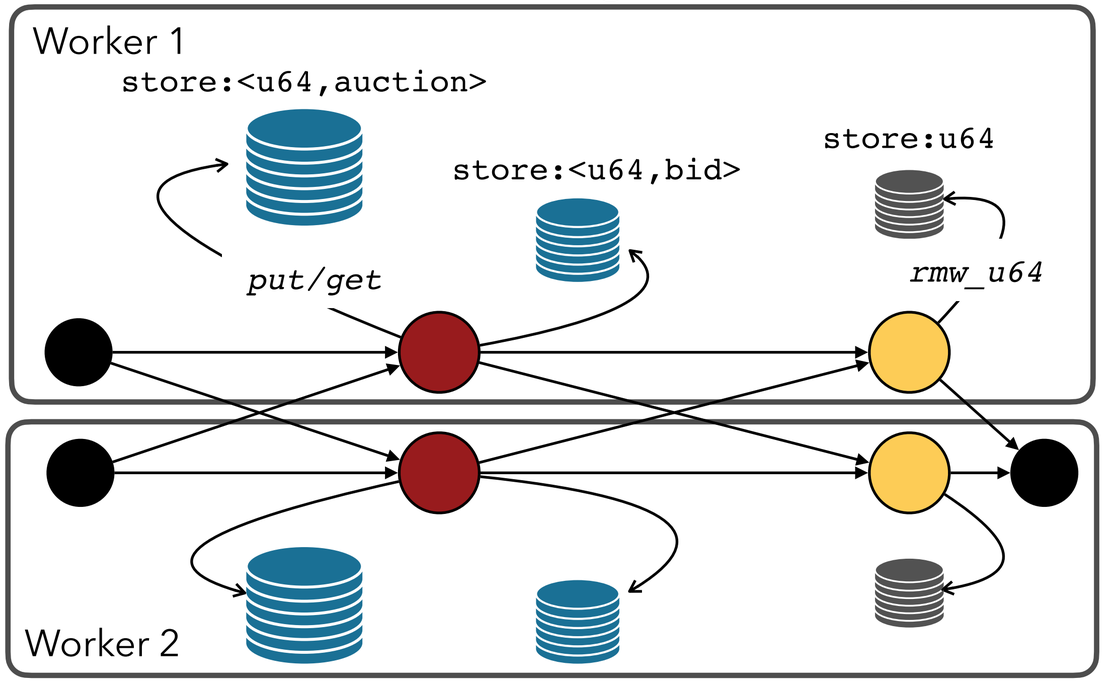
OPEN QUESTIONS

- One store fits all or many?
- Do we need new streaming benchmarks?
- What are the desirable store features to support advanced state operations (e.g. state migration, etc.)?
- How can we learn streaming state characteristics?

SUMMARY

BOSTON UNIVERSITY

Workload-aware streaming state management



- We need to revisit current monolithic approaches
- State store layout affects query performance significantly
- Workload-aware state management achieves up to 14X speedup and 2X higher throughput in Nexmark queries

Testbed: **https://github.com/jliagouris/wassm**

Vasiliki Kalavri
vkalavri@bu.edu

John Liagouris
liagos@bu.edu