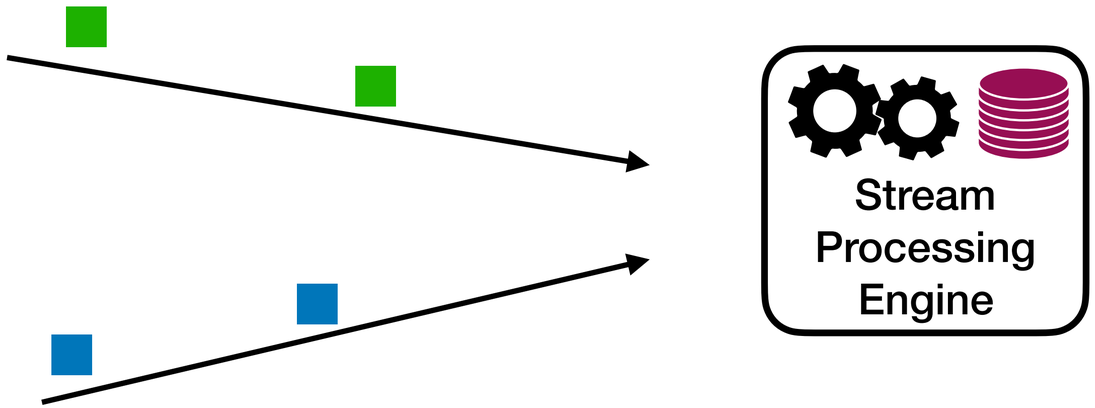
Hardware-Conscious Techniques for Efficient and Reliable Stateful Stream Processing

Ph.D. Thesis Defense

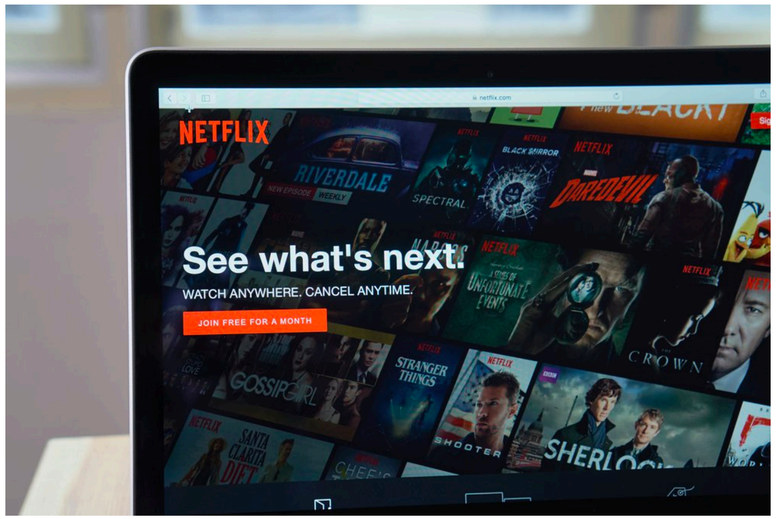
Bonaventura Del Monte

5 December 2022

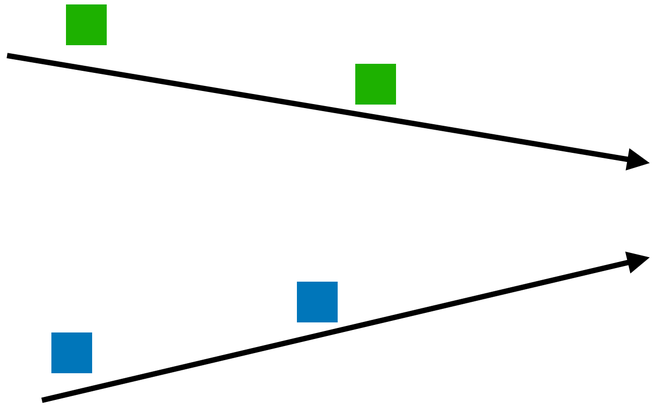
Stateful Streaming Analytics

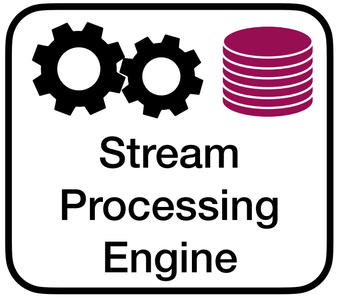


Stateful Streaming Analytics

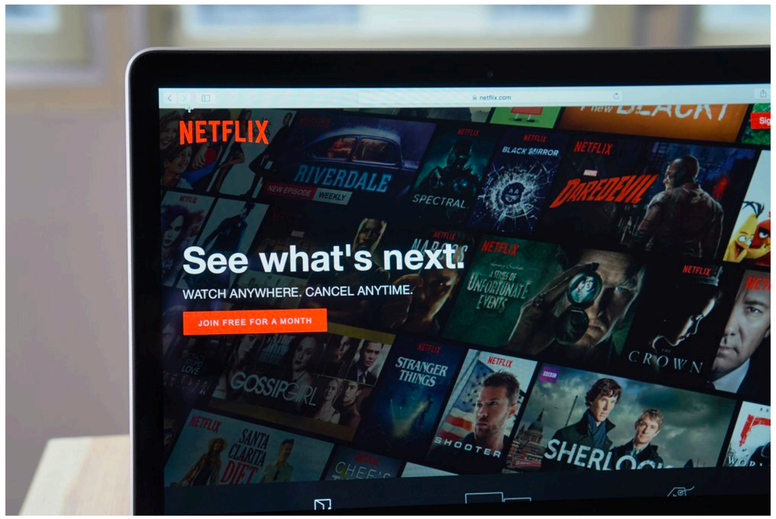


Credit Card Fraud Detection

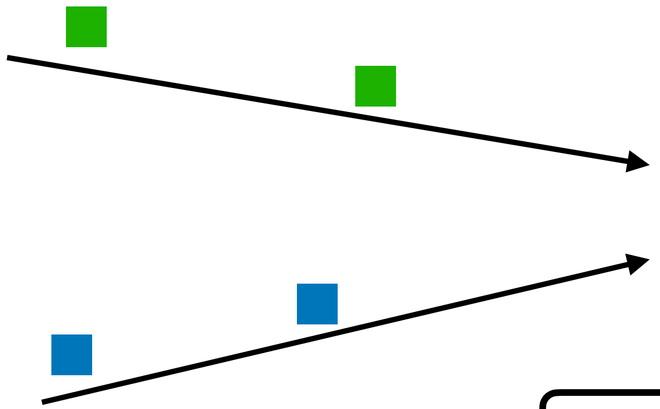


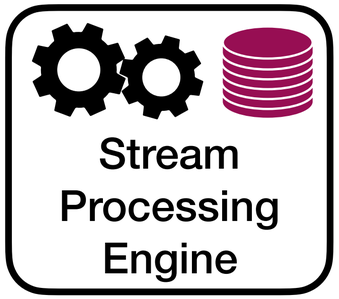


Stateful Streaming Analytics



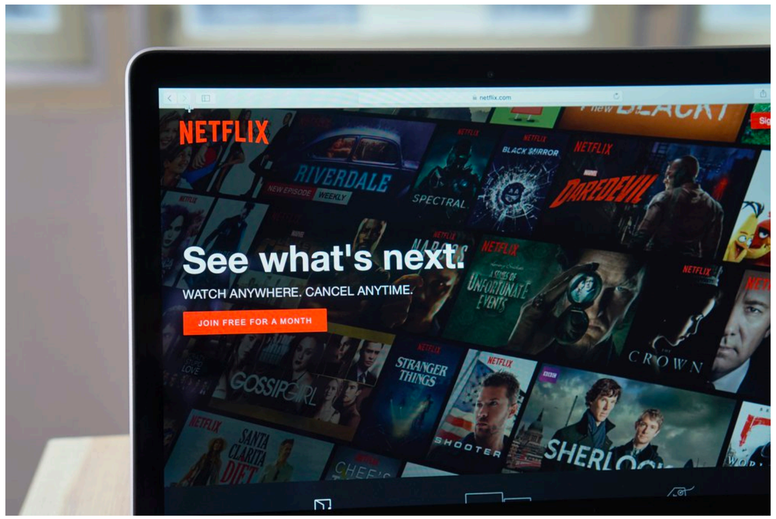
Credit Card Fraud Detection



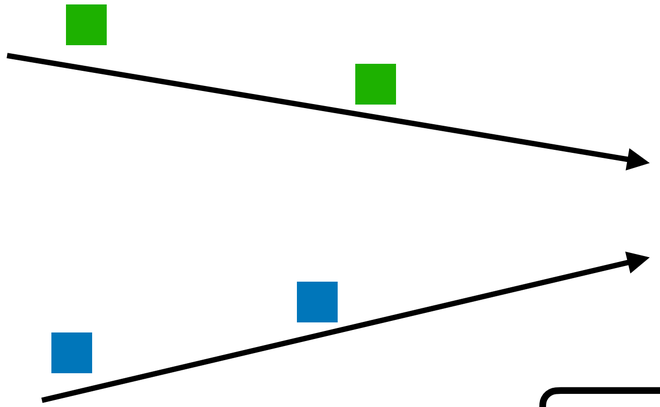


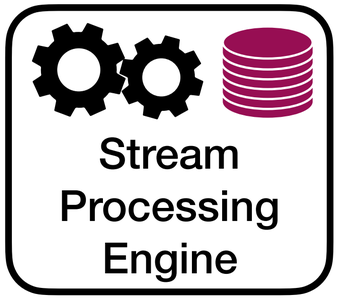
Operator State: mutable dataset of (k,v)

Stateful Streaming Analytics



Credit Card Fraud Detection

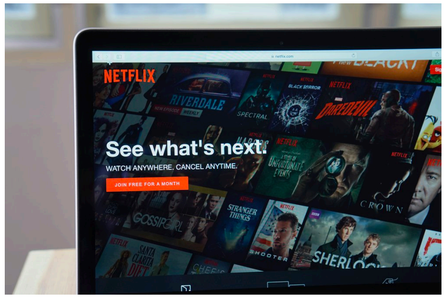




Operator State: mutable dataset of (k,v)

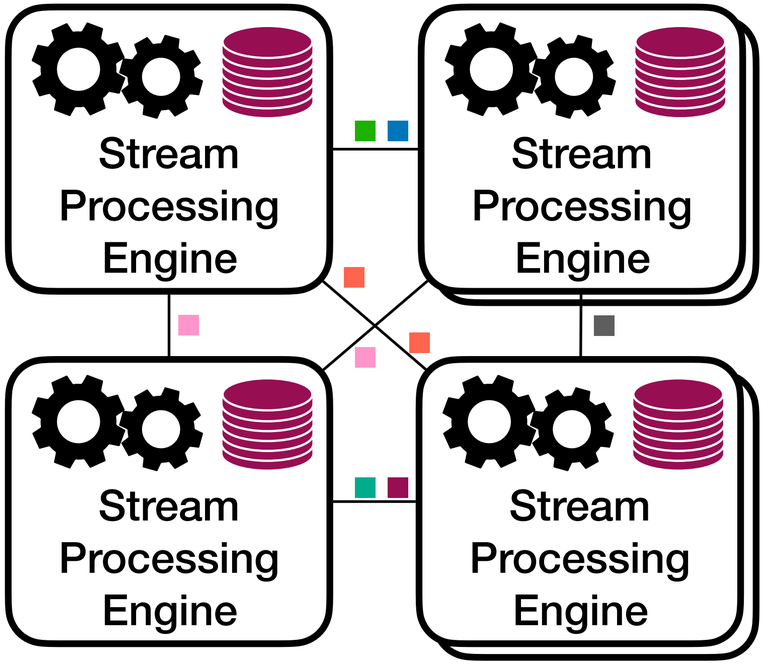
Windowed Aggregations, Windowed Joins, or Machine Learning Tasks

Stateful Streaming Analytics at Scale

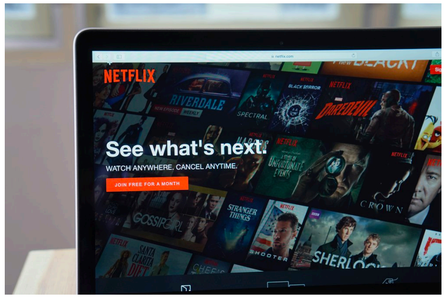


billions of events
high cardinality

Credit Card Fraud Detection



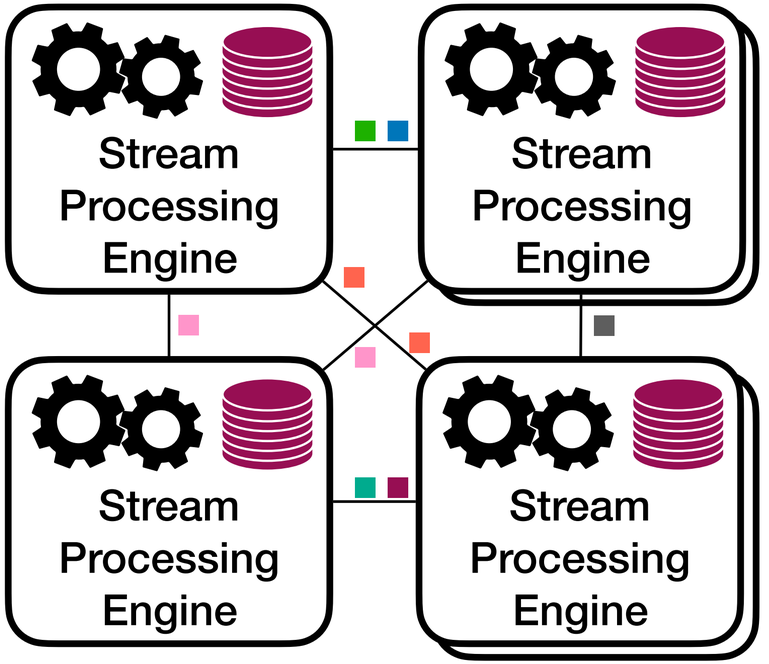
Stateful Streaming Analytics at Scale



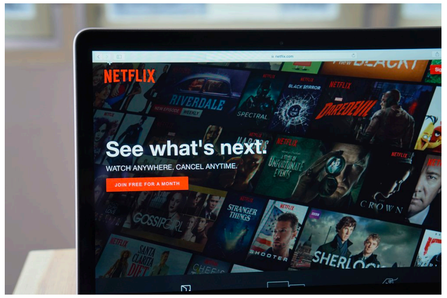
billions of events
high cardinality

Credit Card Fraud Detection

data rate 1-10 **GB/s**



Stateful Streaming Analytics at Scale

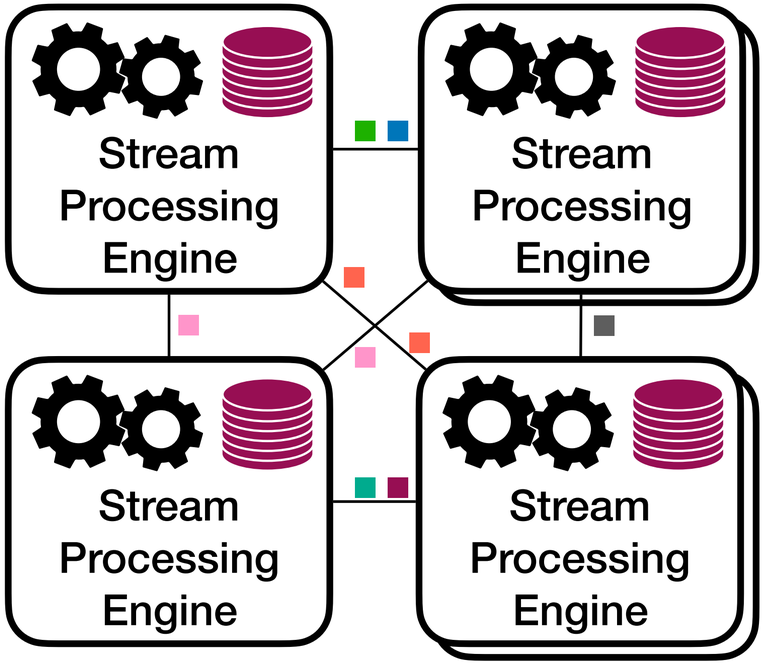


billions of events
high cardinality

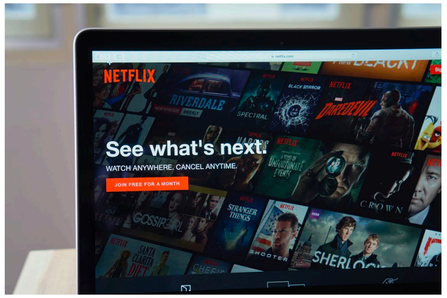
Credit Card Fraud Detection

data rate 1-10 **GB/s**

state size 1-10 **TB**



Stateful Streaming Analytics at Scale

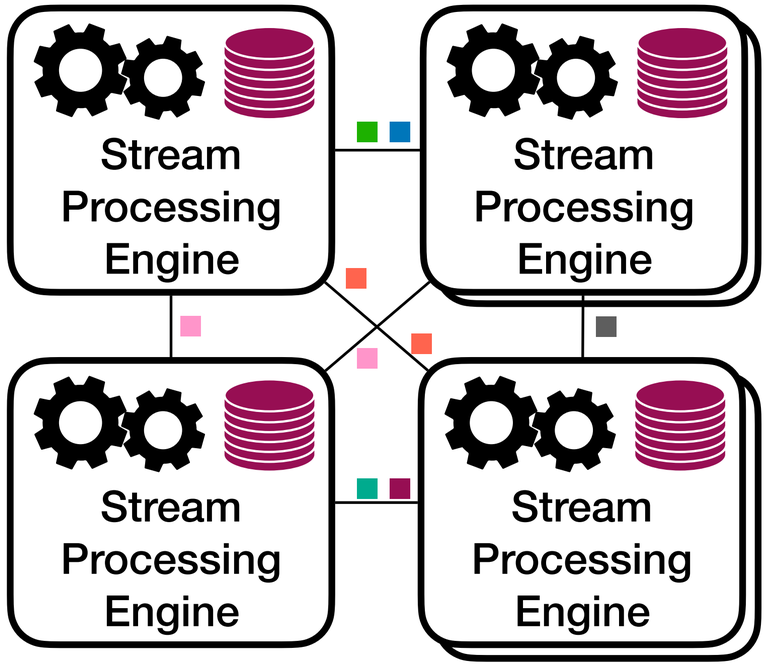


billions of events
high cardinality

Credit Card Fraud Detection

data rate 1-10 **GB/s**

state size 1-10 **TB**

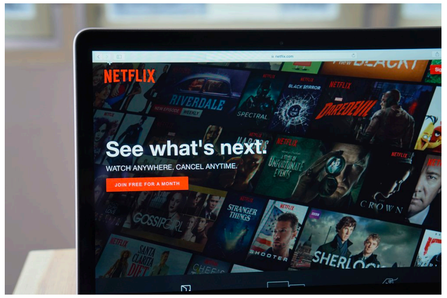


NETFLIX

KLAVIYO

stripe

Stateful Streaming Analytics at Scale

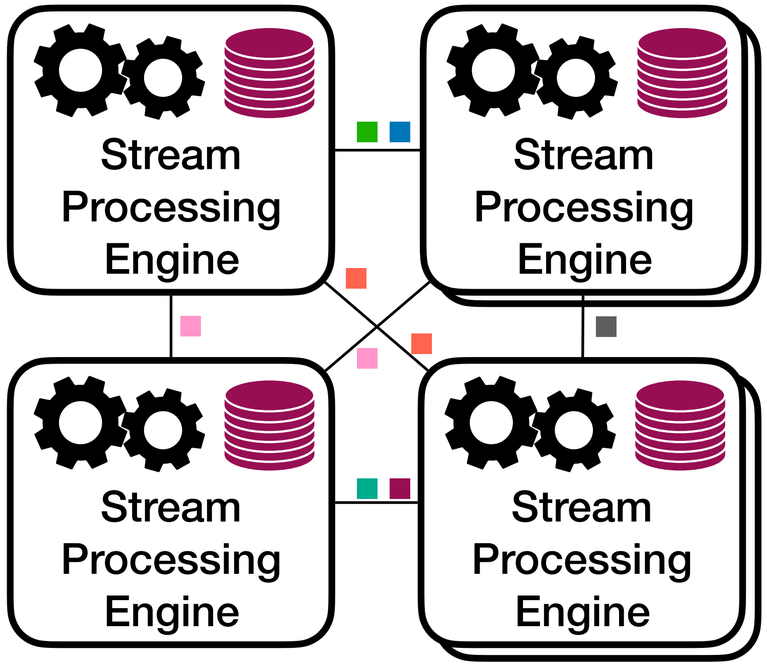


billions of events
high cardinality

data rate 1-10 **GB/s**

Credit Card Fraud Detection

state size 1-10 **TB**



Sustain high-throughput stream processing with low-latency

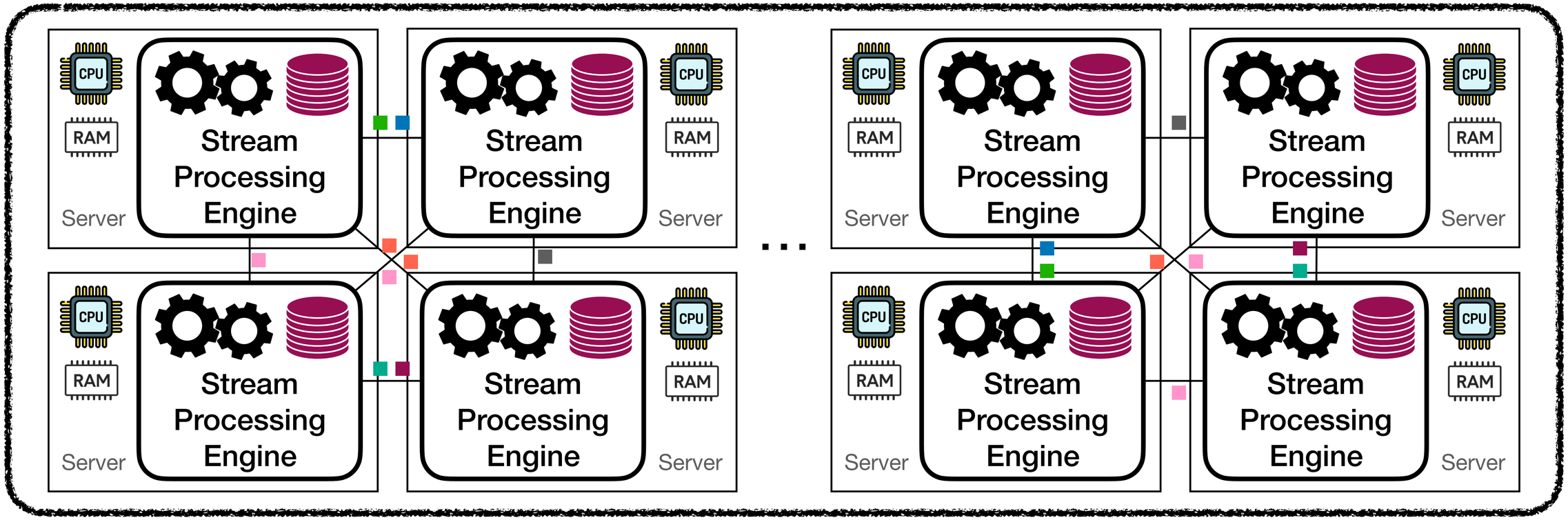
Despite failures and data rate fluctuation

We need efficient and reliable stateful stream processing

Current scale-out SPE architectures

Enable efficient and reliable stateful stream processing

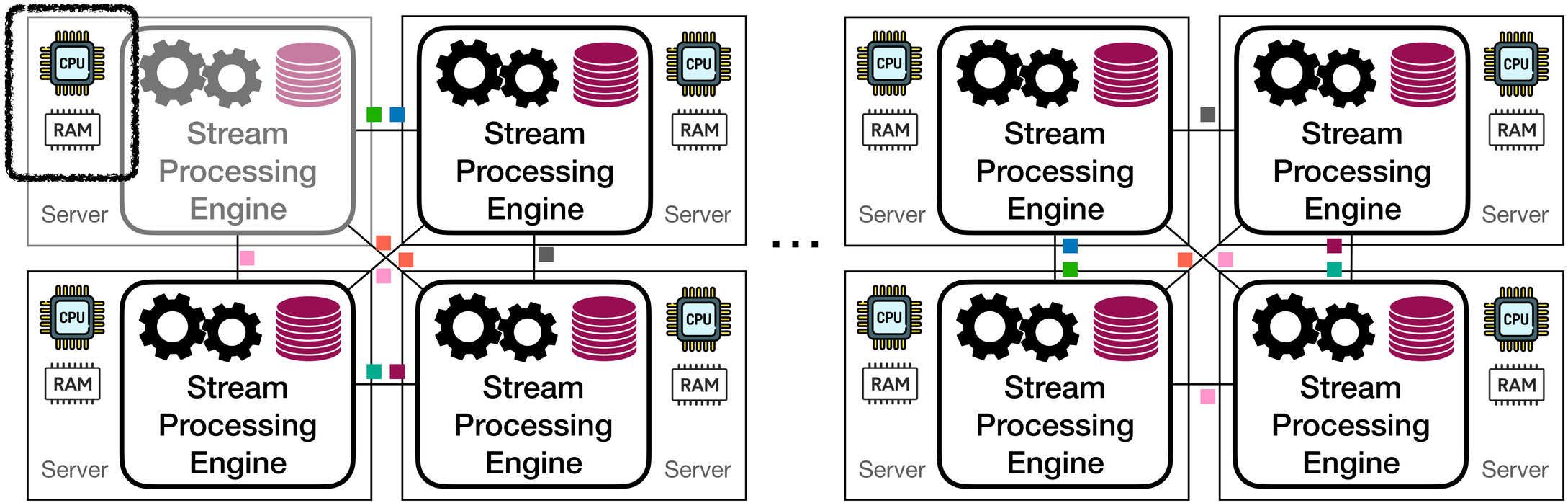
but by scaling out on commodity hardware



Current scale-out SPE architectures

Enable efficient and reliable stateful stream processing

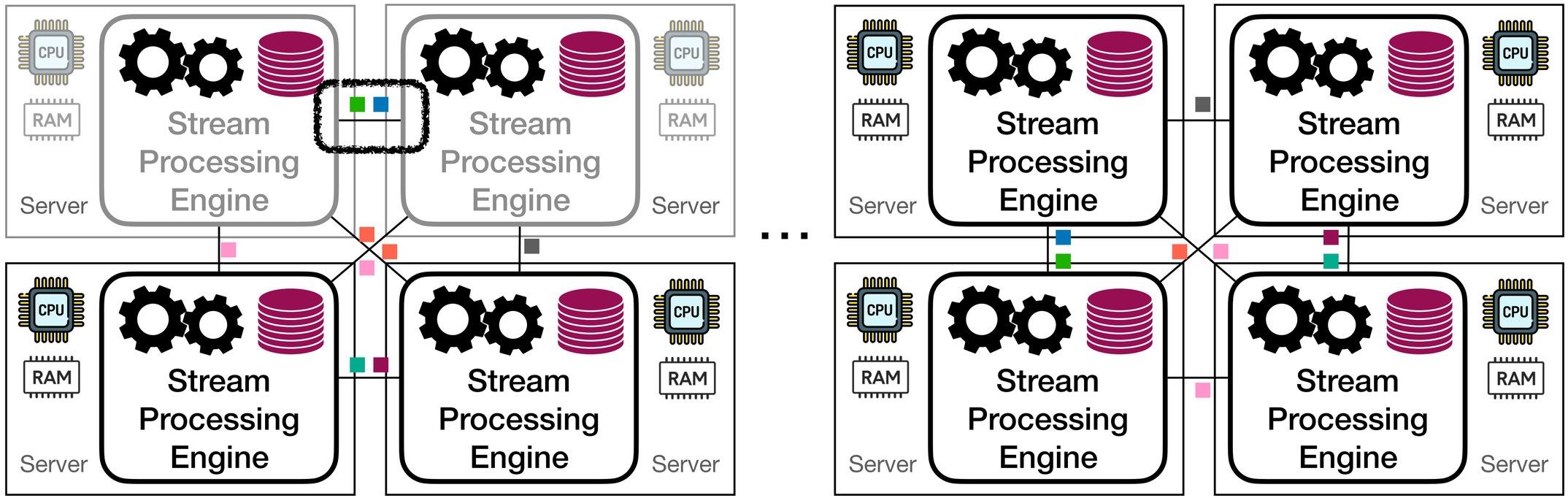
but by scaling out on commodity hardware



Current scale-out SPE architectures

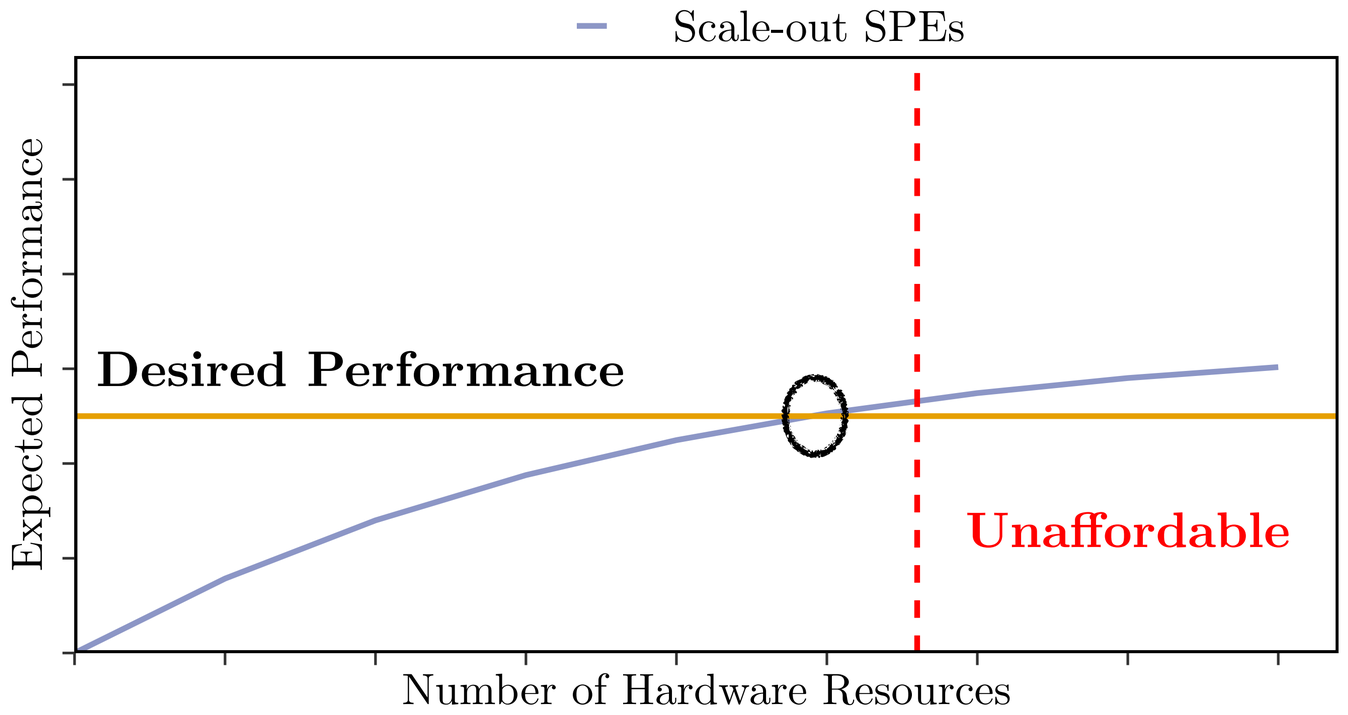
Enable efficient and reliable stateful stream processing

but by scaling out on commodity hardware



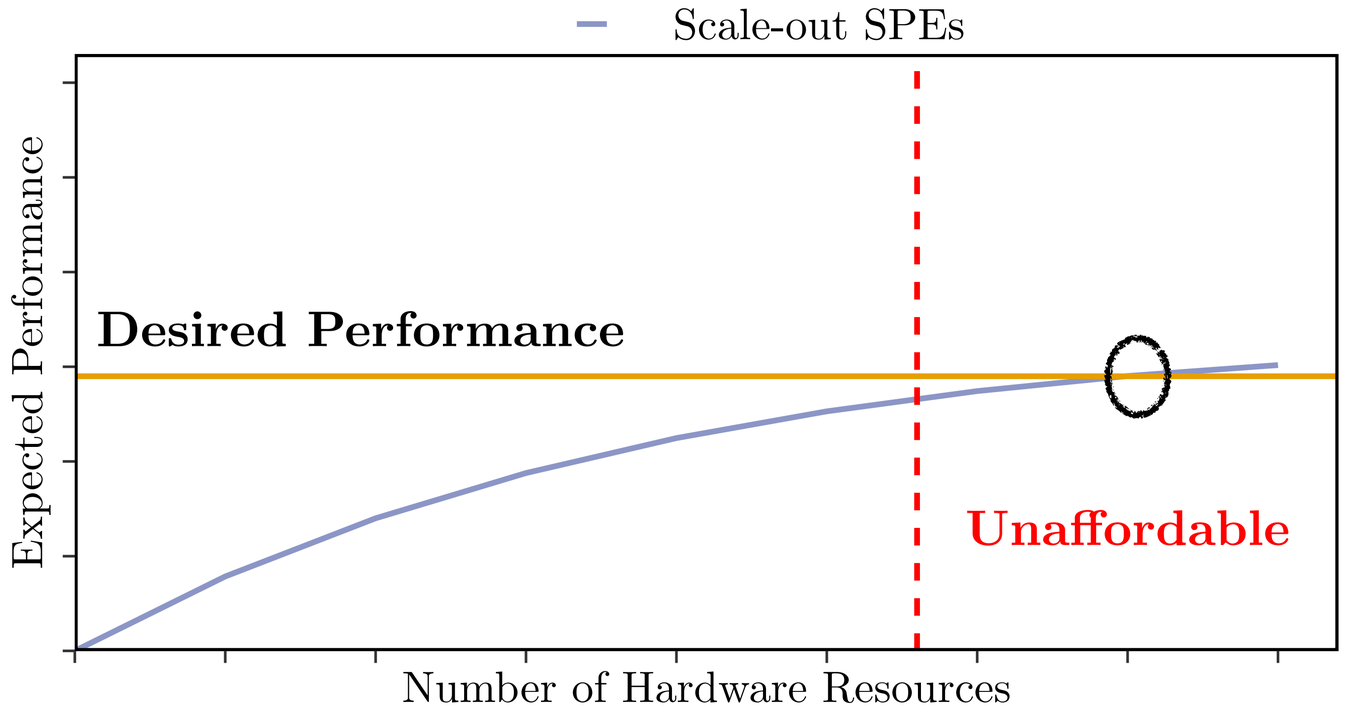
Current scale-out SPE architectures

- Scale-out on commodity hardware
- Add more compute resource to meet desired performance



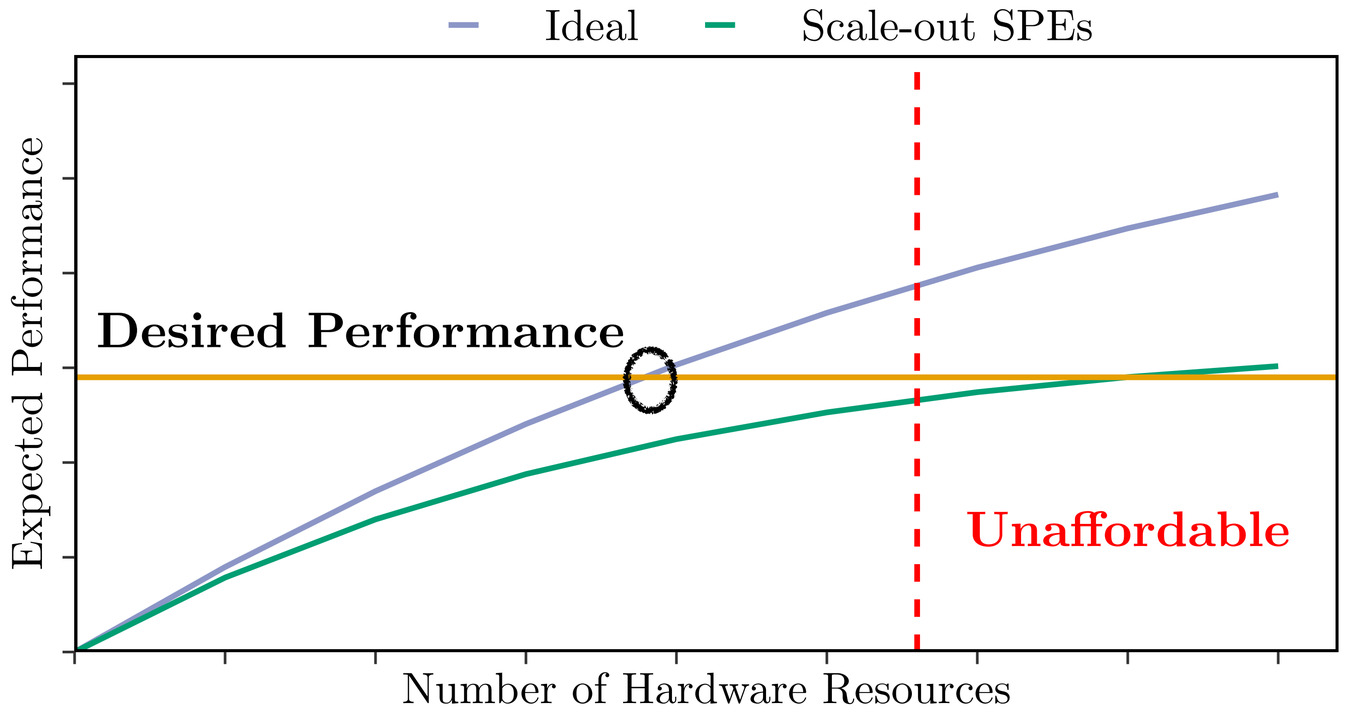
Current scale-out SPE architectures

- Scale-out on commodity hardware
- Add more compute resource to meet desired performance
- Cannot scale out infinitely using finite resources

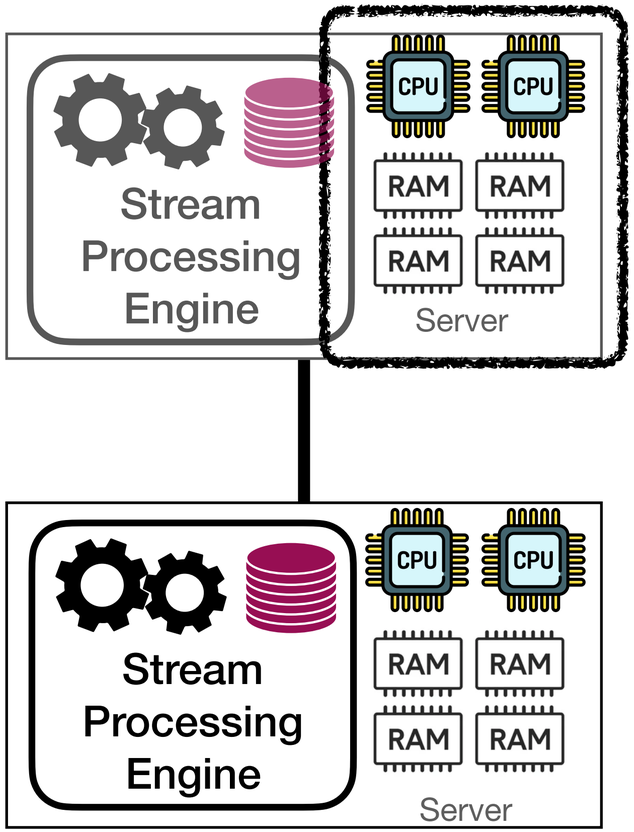


Thesis Goal

Enable efficient and reliable stateful stream processing using hardware more efficiently



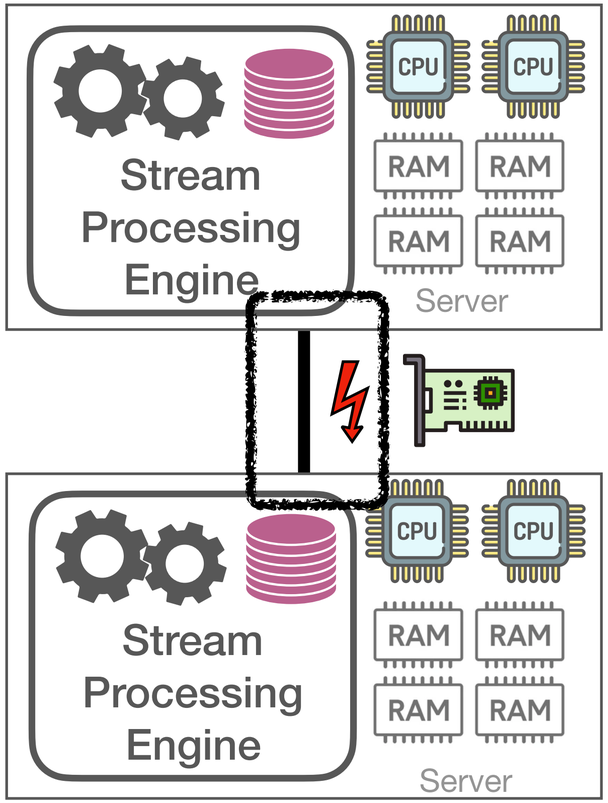
Rethinking the commodity assumption



Compute

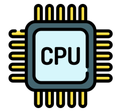
Multi-core CPUs
Large Cache Hierarchy
100s GB Main-Memory

Rethinking the commodity assumption



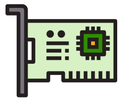
Compute

Multi-core CPUs
Large Cache Hierarchy
100s GB Main-Memory

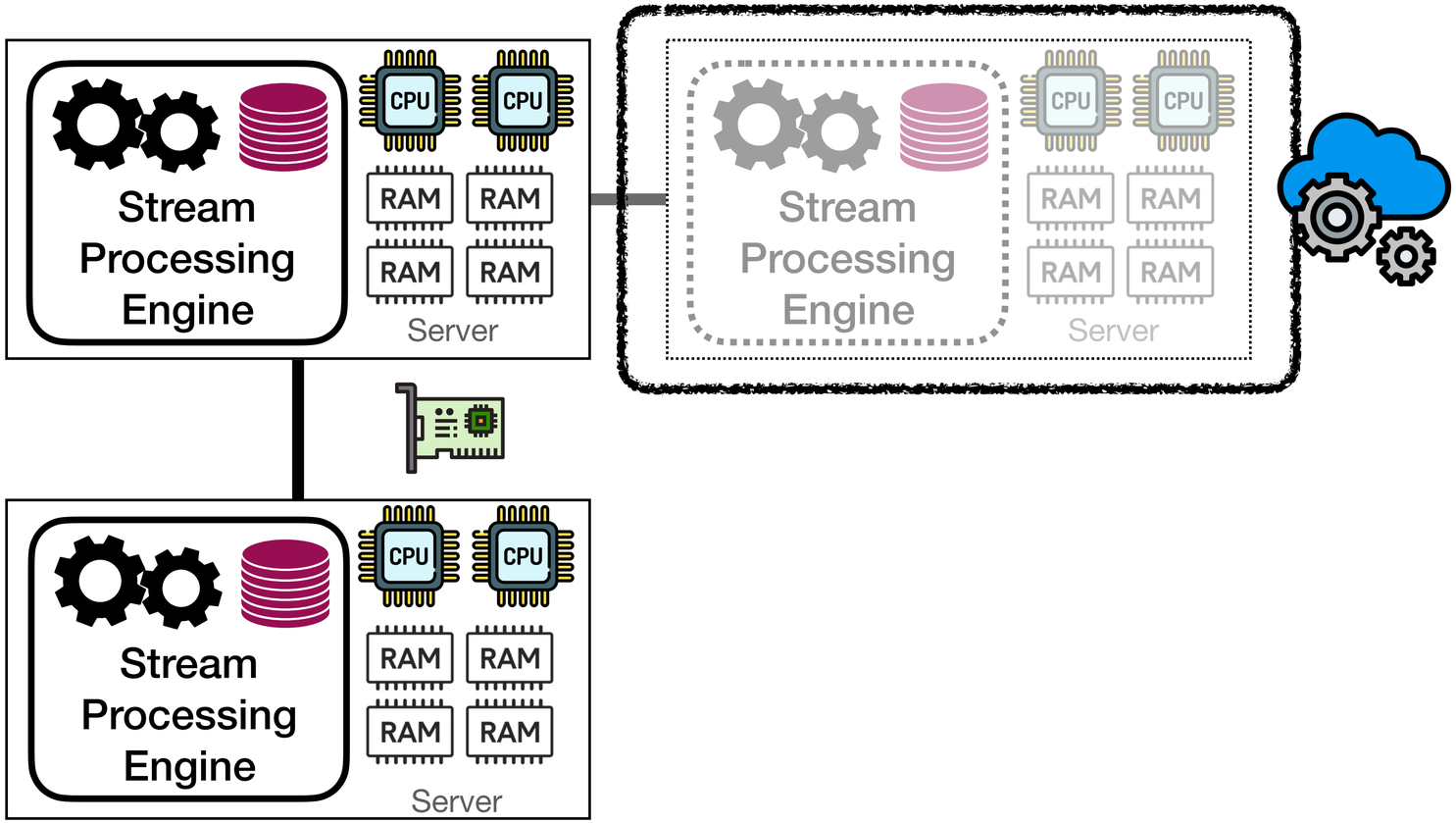


Network

High-speed Networking
Close to memory bandwidth
Faster than 10Gbps Ethernet



Rethinking the commodity assumption



Compute

Multi-core CPUs
Large Cache Hierarchy
100s GB Main-Memory

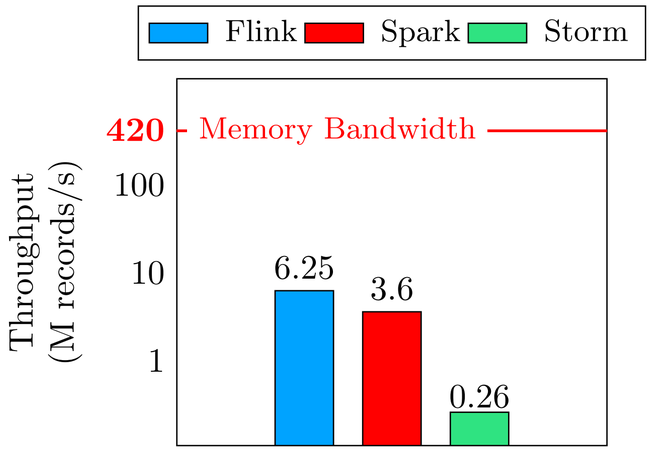
Network

High-speed Networking
Close to memory bandwidth
Faster than 10Gbps Ethernet

Elasticity

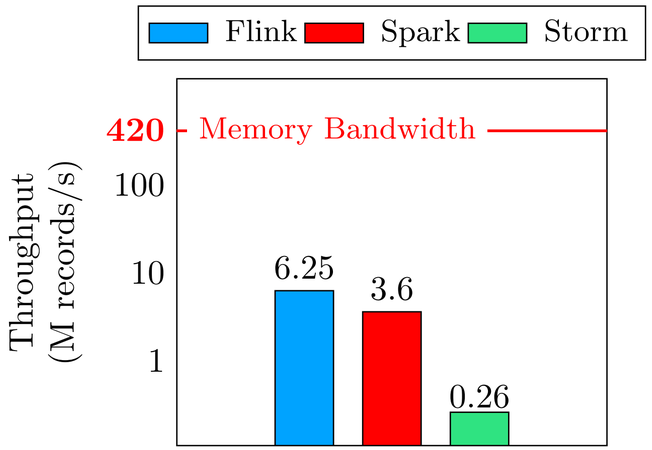
Flexible Provisioning
Reconfiguration

SPEs don’t scale *with* the hardware capabilities

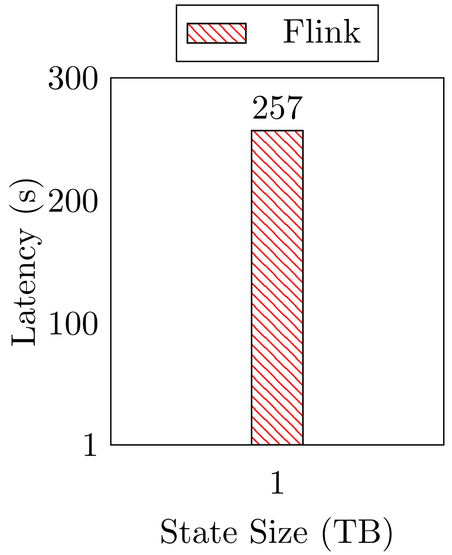


Query Processing

SPEs don’t scale *with* the hardware capabilities

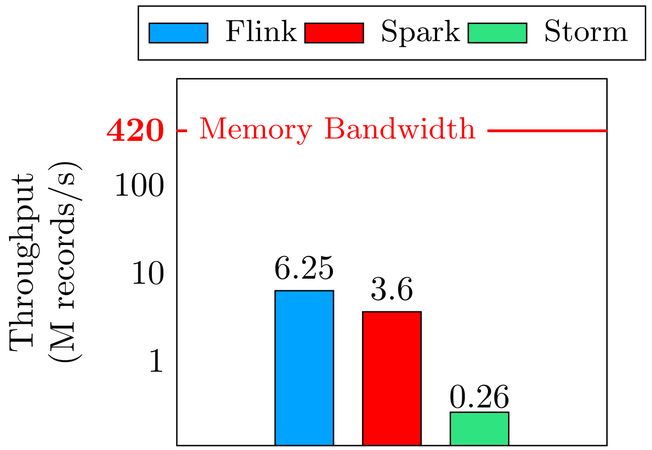


Query Processing

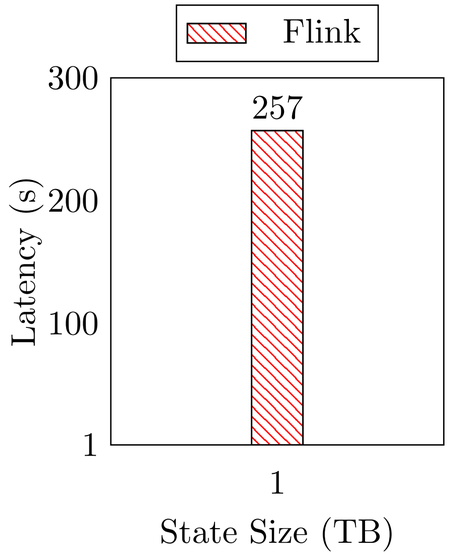


Query Reconfiguration

SPEs don’t scale *with* the hardware capabilities



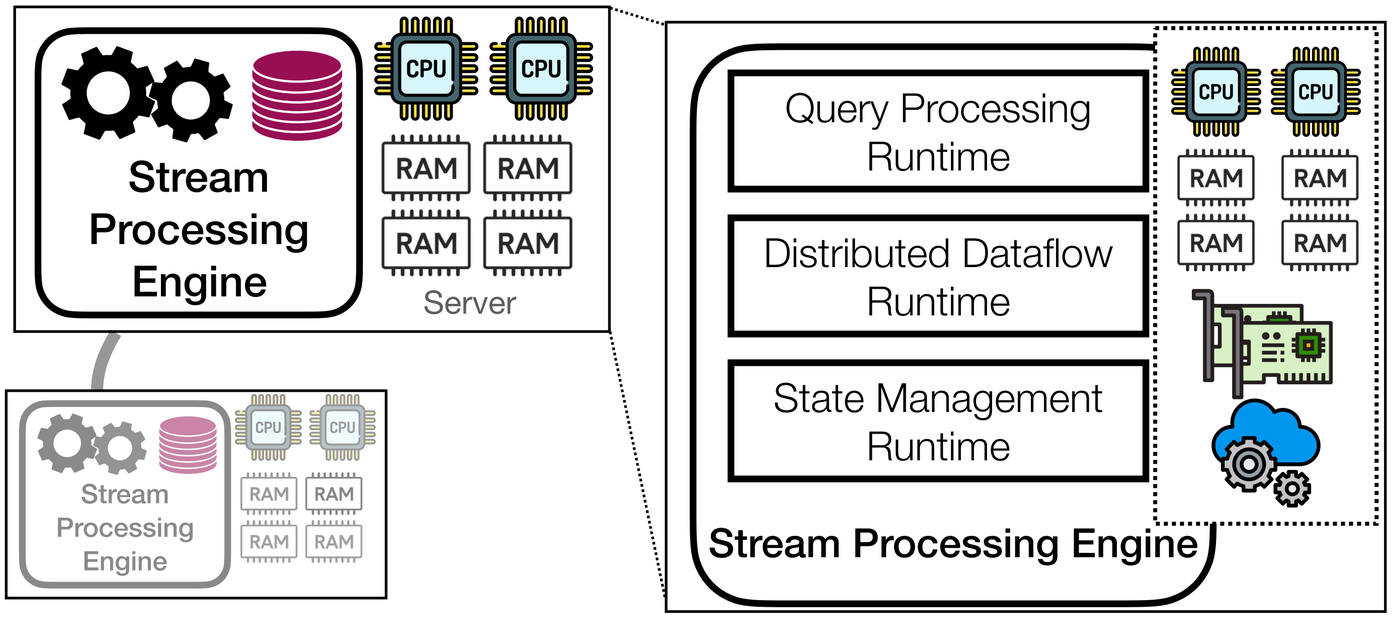
Query Processing



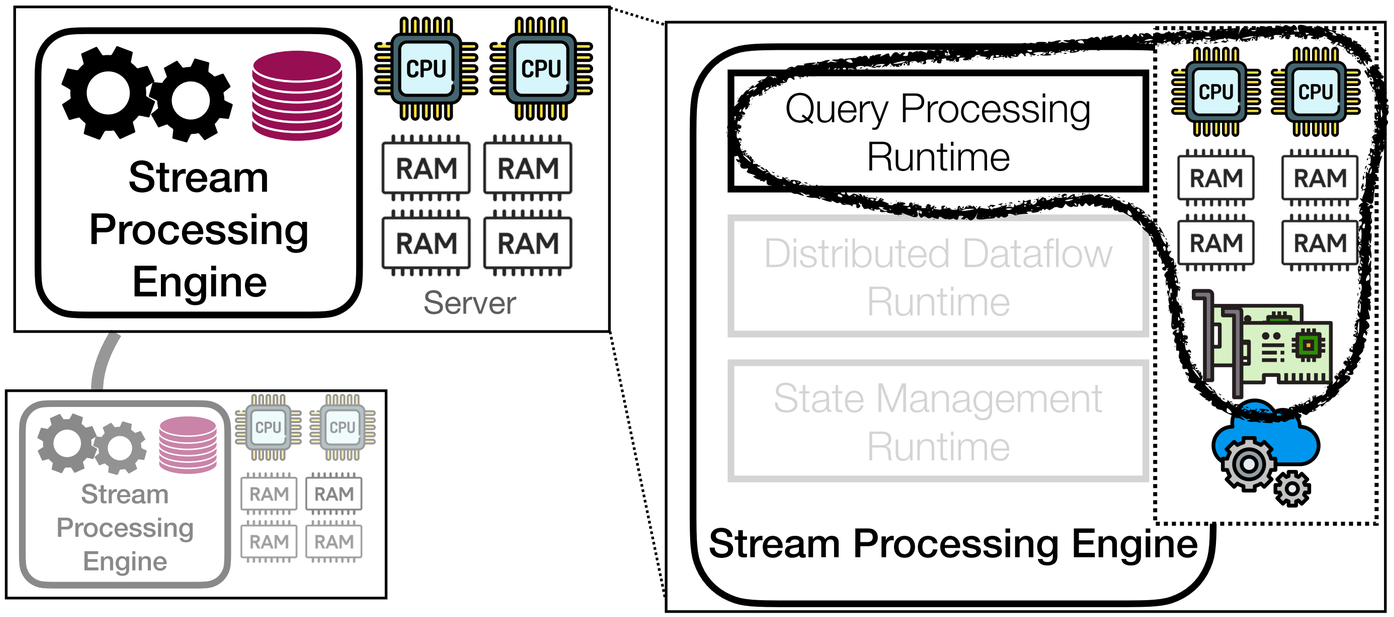
Query Reconfiguration

Problem: Hardware-oblivious SPE design does not enable efficient and reliable stateful stream processing

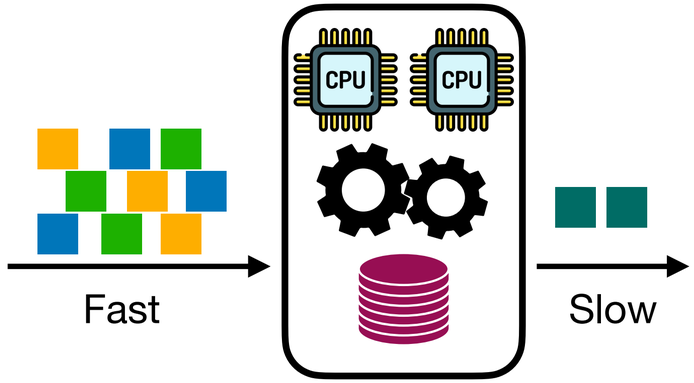
Challenges: designing hardware-conscious SPEs



Challenges: designing hardware-conscious SPEs



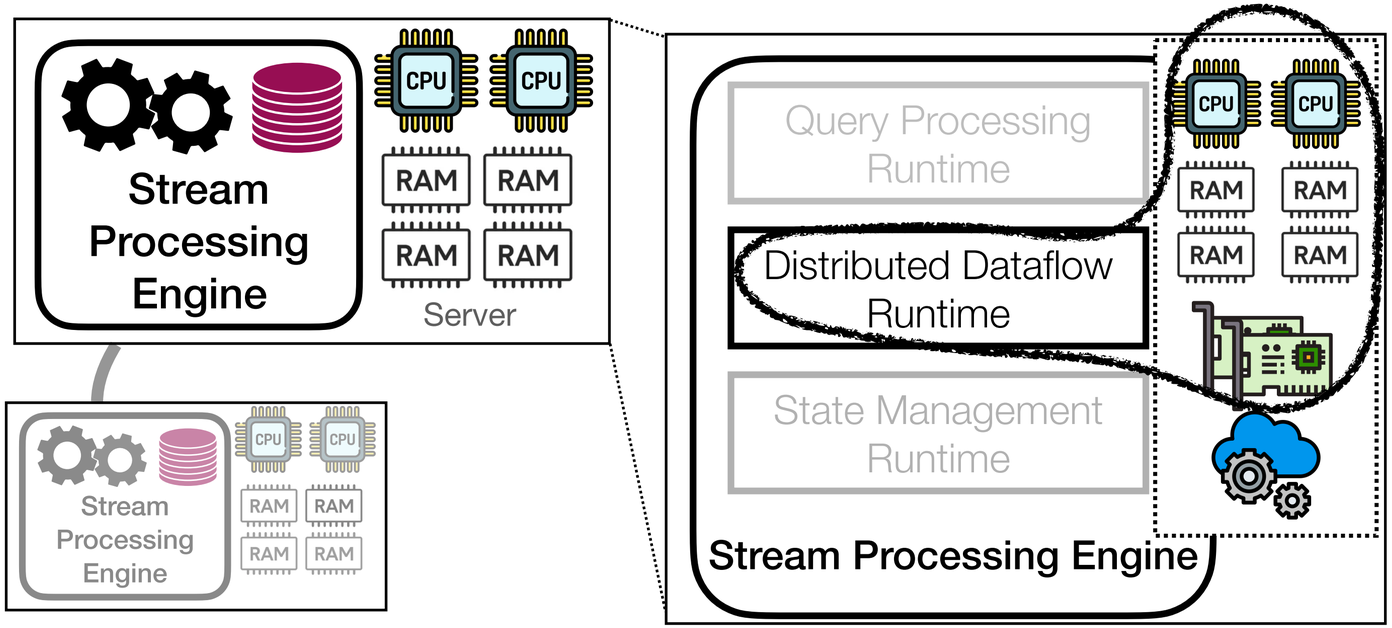
SPEs are CPU-Bound with high-speed networks



Inefficient Query Execution

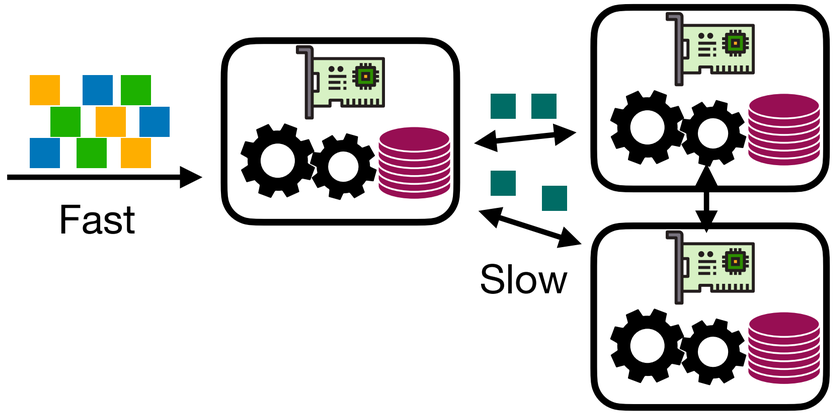
Inefficient Memory Access Patterns

Challenges: designing hardware-conscious SPEs



SPEs are CPU-Bound with high-speed networks

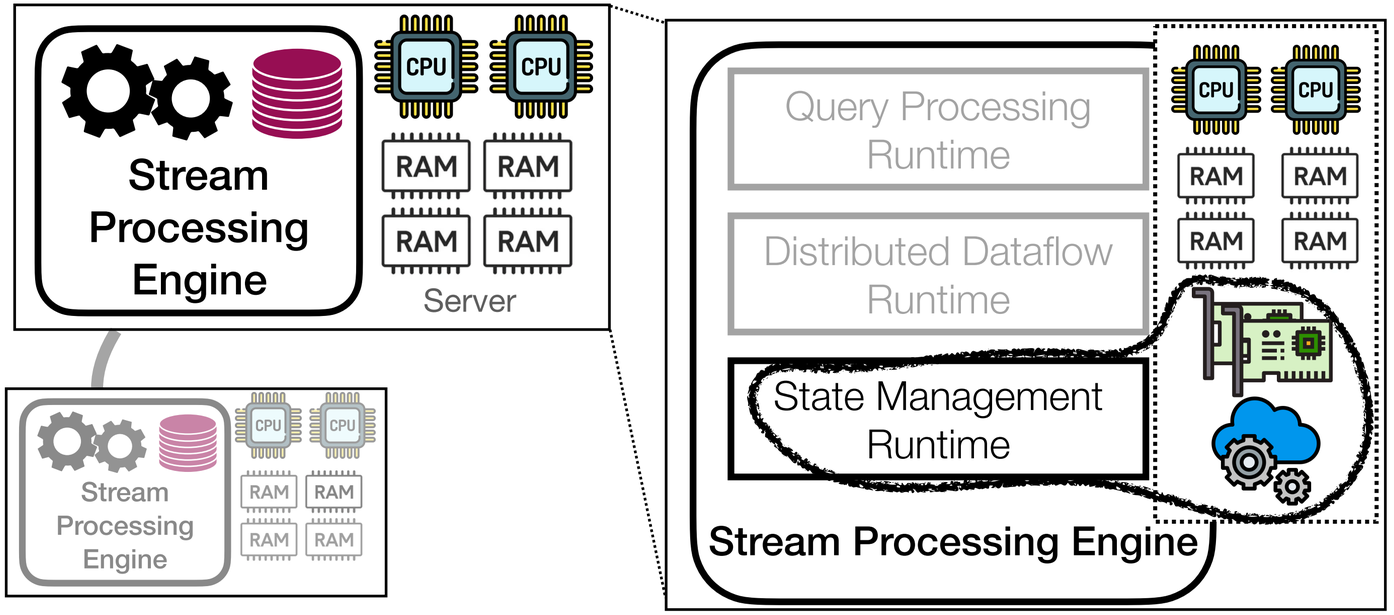
SPEs cannot fully use high-speed networks to scale-out



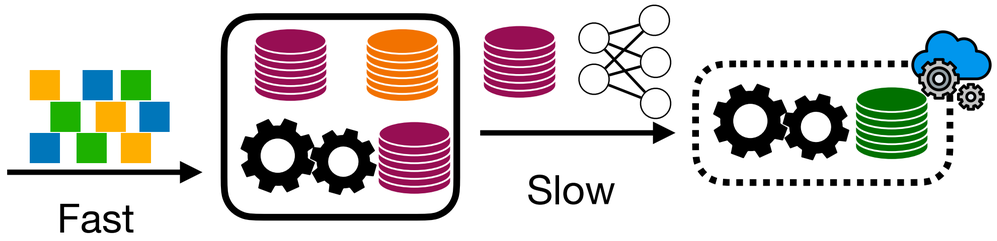
Expensive Data Shuffling

OS-Managed Networking

Challenges: designing hardware-conscious SPEs



Large state causes a bottleneck for on-the-fly query reconfiguration



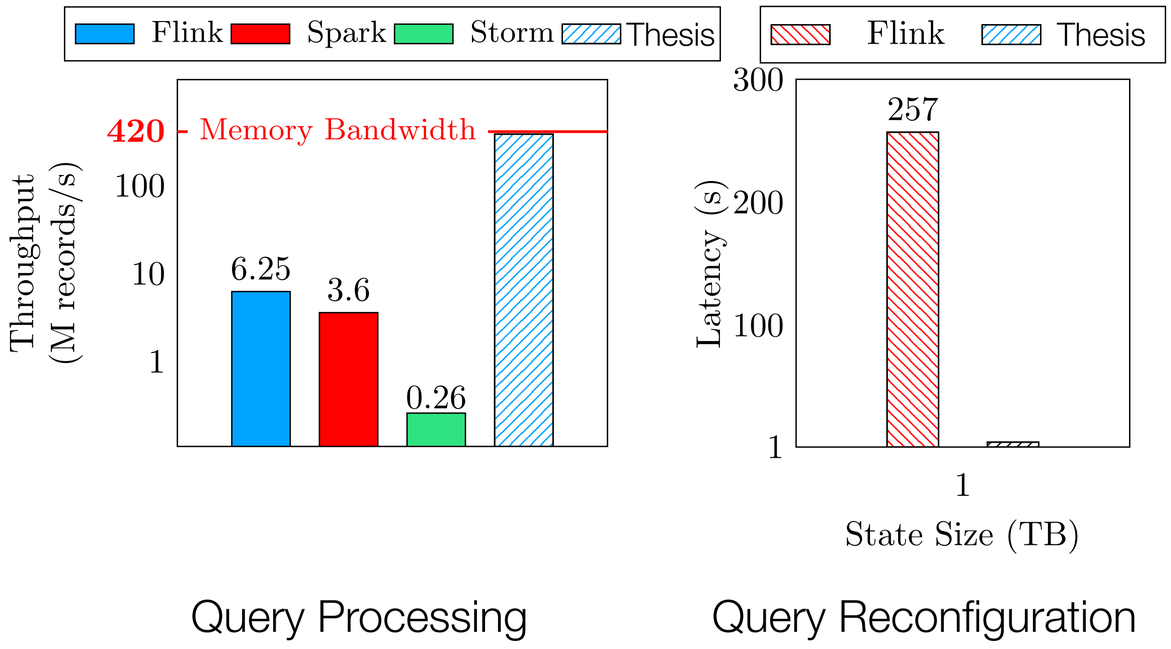
SPEs are CPU-Bound with high-speed networks

SPEs cannot fully use high-speed networks

Processing & Network Overhead

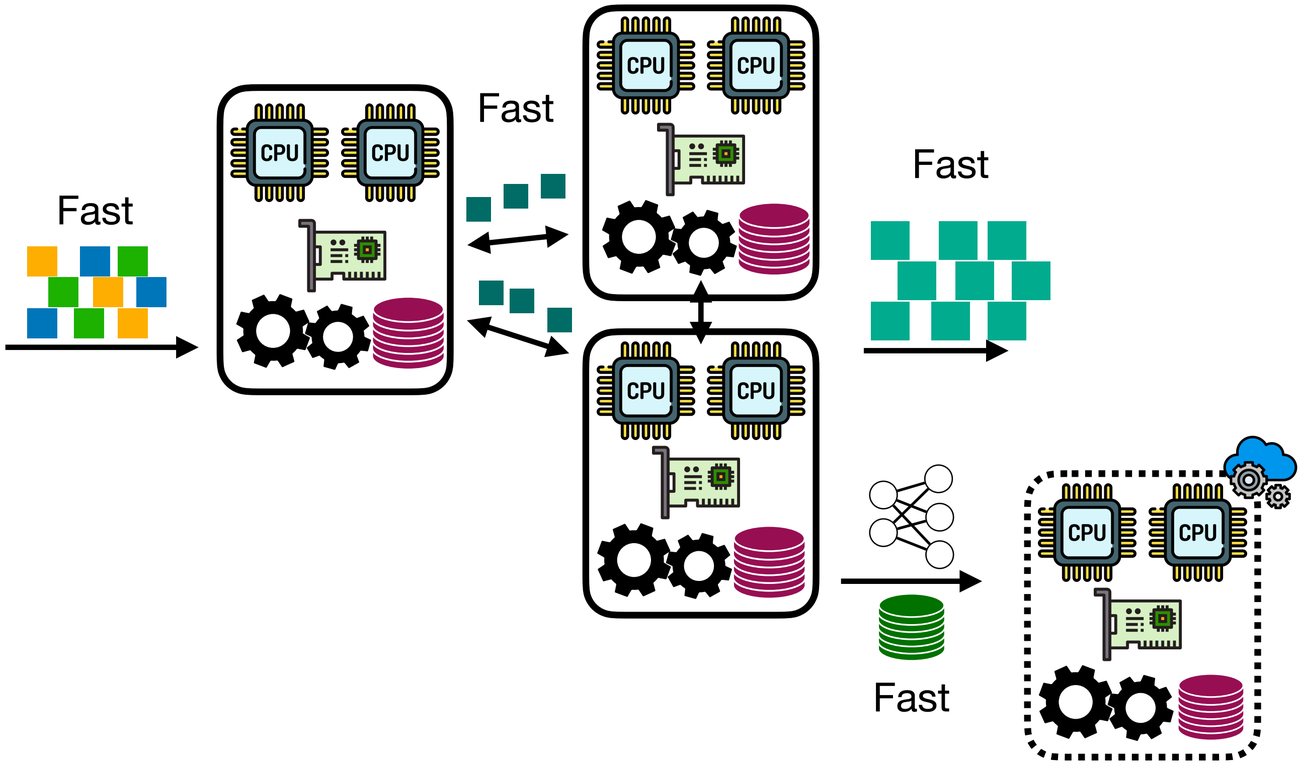
Dataflow Computation Consistency

Thesis Solution



Query Processing

Query Reconfiguration



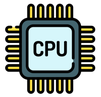
Adopt hardware-conscious SPE design to enable efficient and reliable stateful stream processing

Contributions: Hardware-conscious techniques for SPEs

SPEs are CPU-Bound with high-speed networks

Understand Stream Processing Performance on Modern Hardware

E&A PVLDB 2019

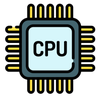


Contributions: Hardware-conscious techniques for SPEs

SPEs are CPU-Bound with high-speed networks

Understand Stream Processing Performance on Modern Hardware

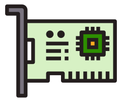
E&A PVLDB 2019



SPEs cannot fully use high-speed networks to scale-out

Efficient Scale-out Processing with High-speed Networks

SIGMOD 2022

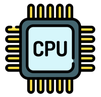


Contributions: Hardware-conscious techniques for SPEs

SPEs are CPU-Bound with high-speed networks

Understand Stream Processing Performance on Modern Hardware

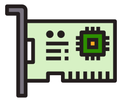
E&A PVLDB 2019



SPEs cannot fully use high-speed networks to scale-out

Efficient Scale-out Processing with High-speed Networks

SIGMOD 2022



Large state is a bottleneck for on-the-fly query reconfiguration

Reliable Management of Very Large Operator State

SIGMOD 2020

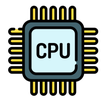
Agenda

SPEs are CPU-Bound with high-speed networks

Understand Stream Processing Performance on Modern Hardware

E&A PVLDB 2019

1

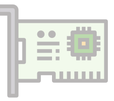


SPEs cannot fully use high-speed networks to scale-out

Efficient Scale-out Processing with High-speed Networks

SIGMOD 2022

2



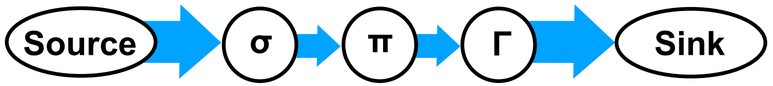
Large state is a bottleneck for on-the-fly query reconfiguration

Reliable Management of Very Large Operator State

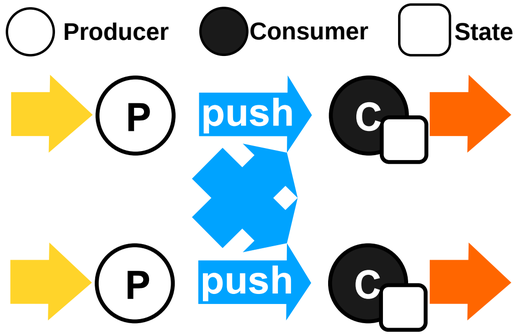
SIGMOD 2020

3

Executing streaming queries on SPEs

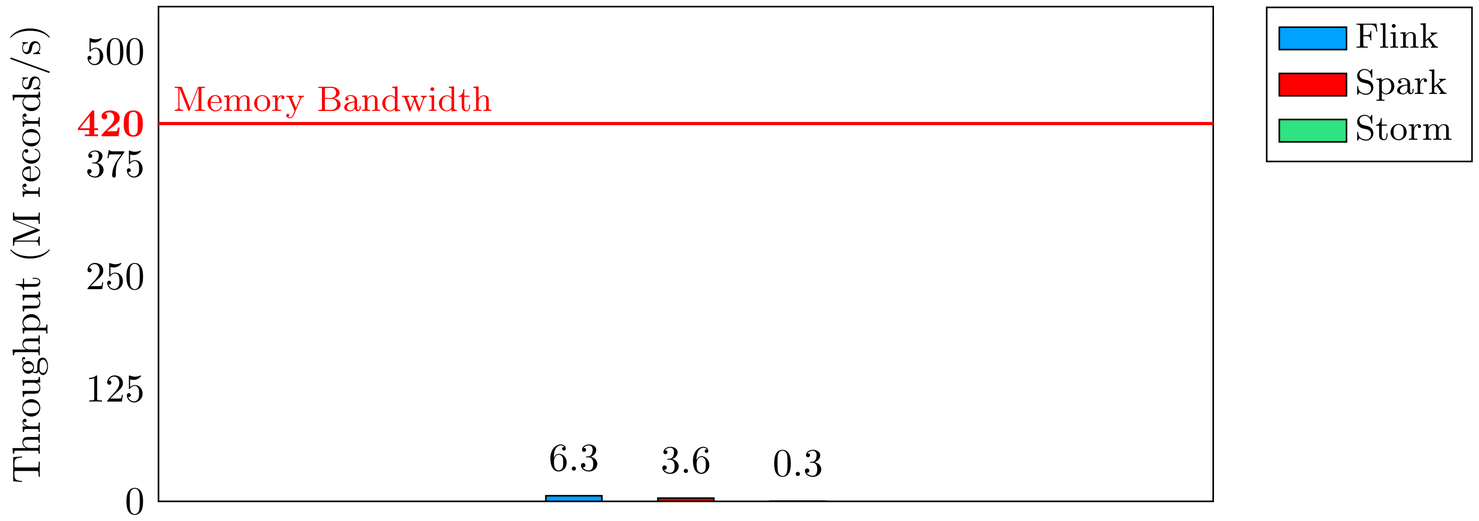


Interpretation-based Query Execution



Upfront Partitioning

Executing streaming queries on SPEs

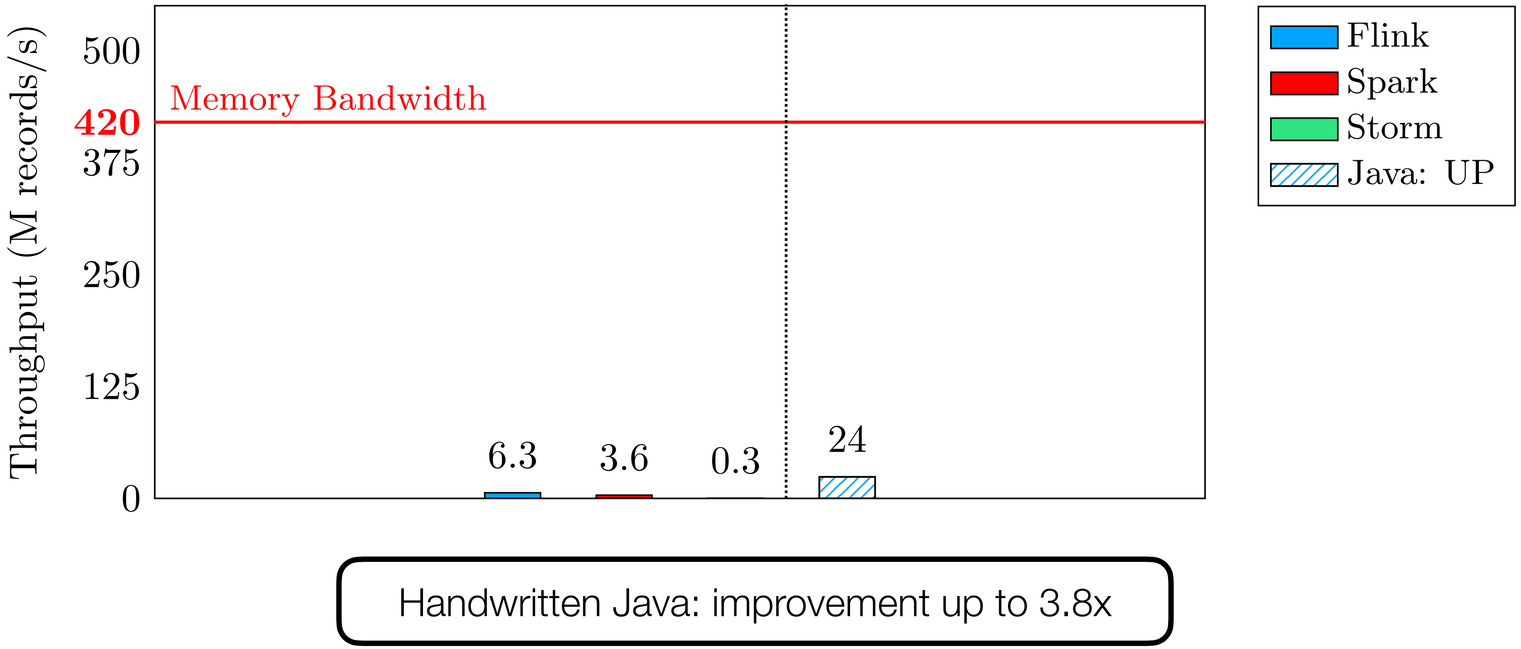


Main-memory as “fast-network”

Yahoo! Streaming Benchmark

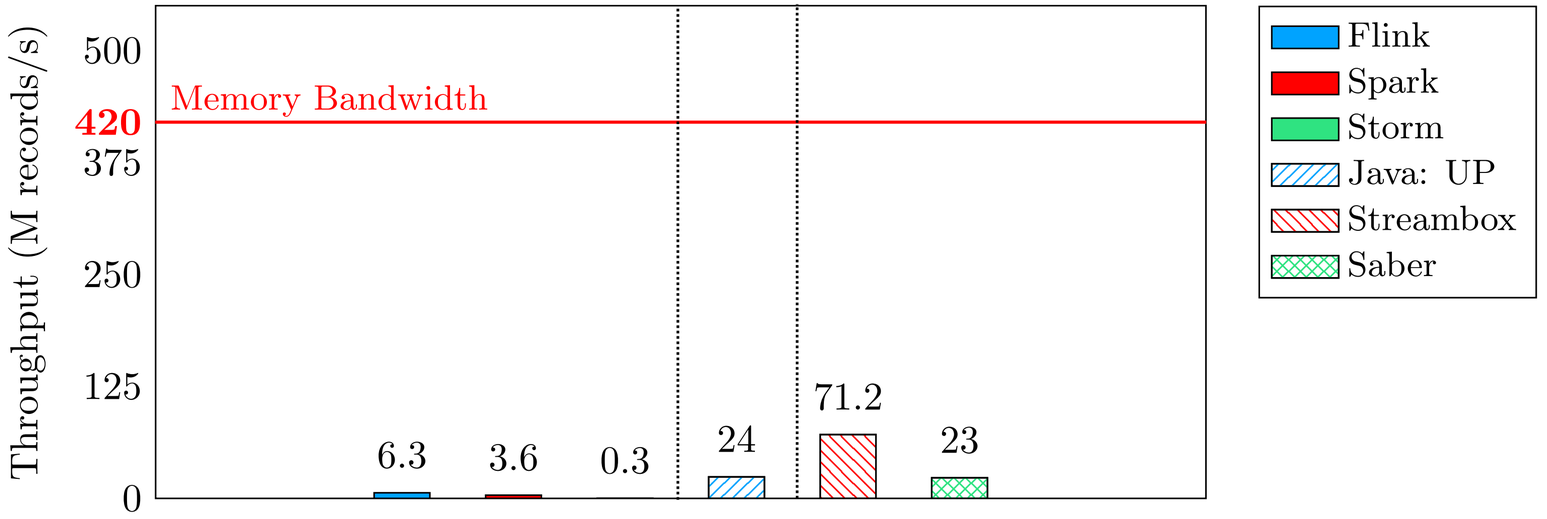
Intel i7 6700K @ 4 Ghz
L1: 32KB L2: 256 KB L3: 8MB
RAM: 32 GB

Executing streaming queries on SPEs



Handwritten Java: improvement up to 3.8x

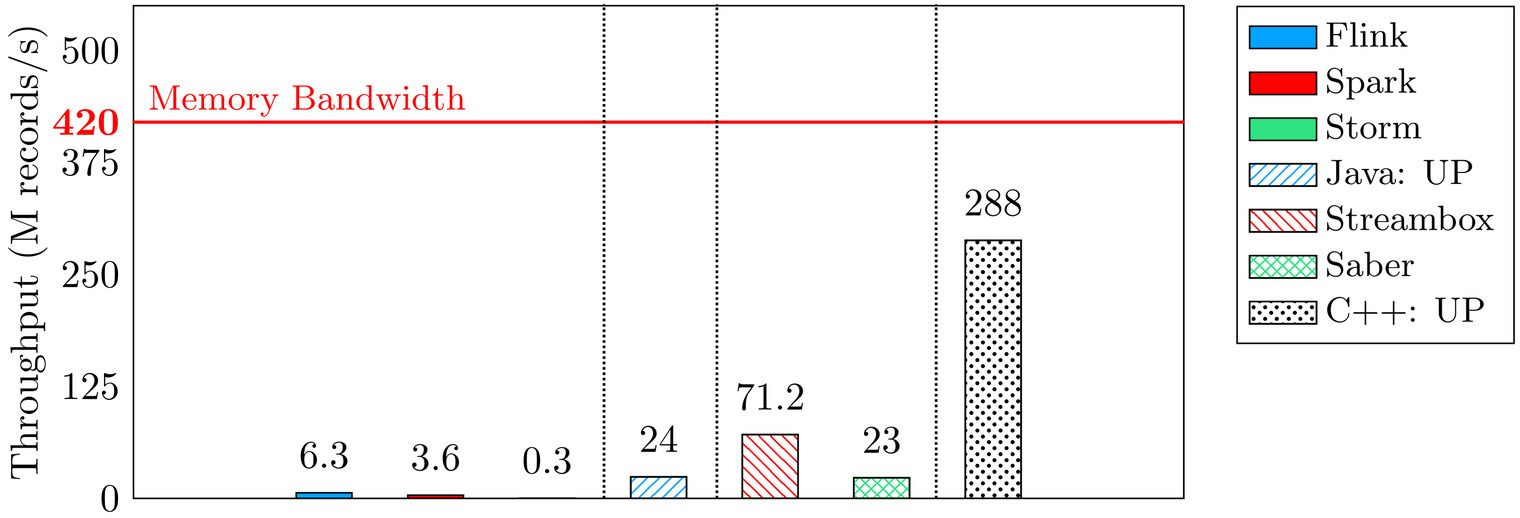
Executing streaming queries on SPEs



Handwritten Java: improvement up to 3.8x

Existing scale-up SPE: improvement up to 3x

Executing streaming queries on SPEs



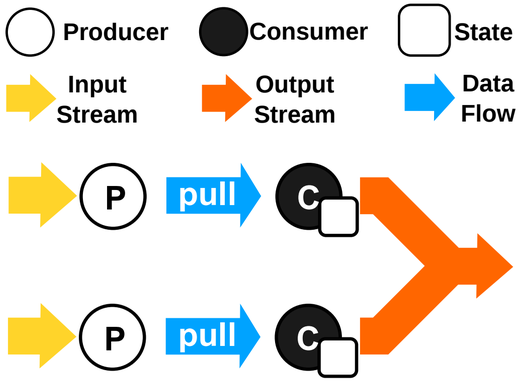
Handwritten Java: improvement up to 3.8x

Existing scale-up SPE: improvement up to 3x

Handwritten C++: improvement up to 4x

Proposed changes for *scale-up* design

Compute and merge partial states without data shuffling

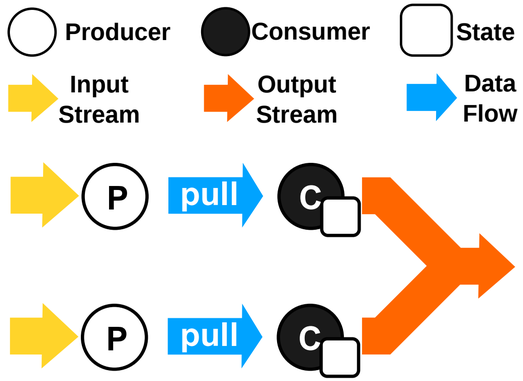


Late Merge (LM)

Operator Parallelization strategies

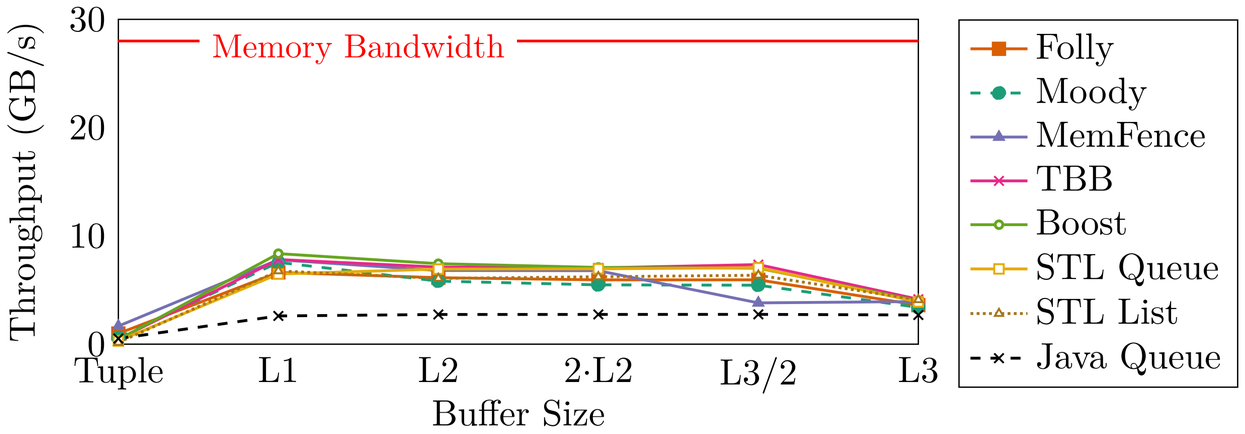
Proposed changes for *scale-up* design

Compute and merge partial states without data shuffling



Late Merge (LM)

Operator Parallelization strategies

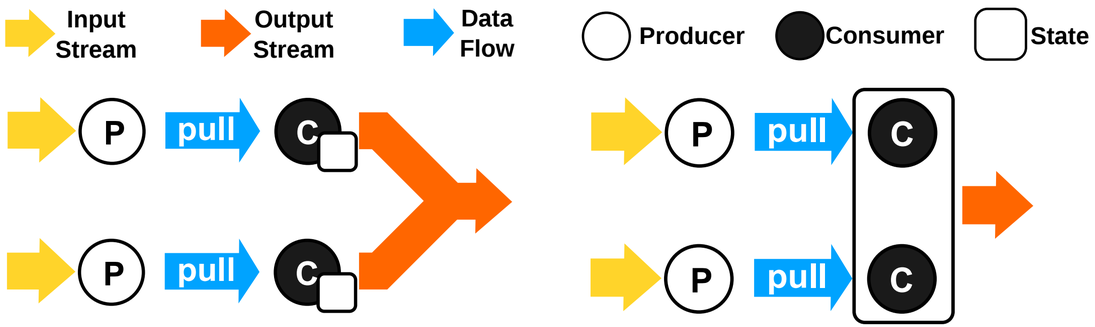


Upfront Partitioning using queues does not achieve full bandwidth even when batching

Proposed changes for *scale-up* design

Compute and merge partial states without data shuffling

Parallelization: GM or LM instead of Upfront Partitioning



Late Merge (LM)

Global Merge (GM)

Operator Parallelization strategies

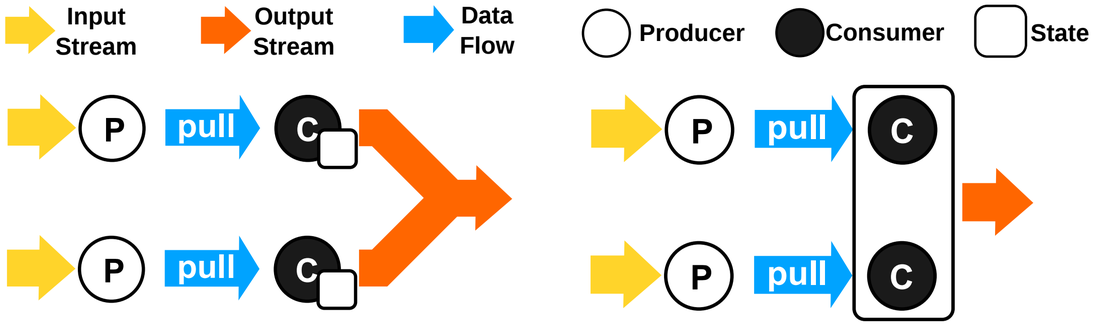
Proposed changes for *scale-up* design

Compute and merge partial states without data shuffling

Enable efficient data passing and code invocation

Parallelization: GM or LM instead of Upfront Partitioning

Hardware-tailored Query Compilation for Stream Processing



Late Merge (LM)

Global Merge (GM)

Operator Parallelization strategies

Compilation-based Query Execution

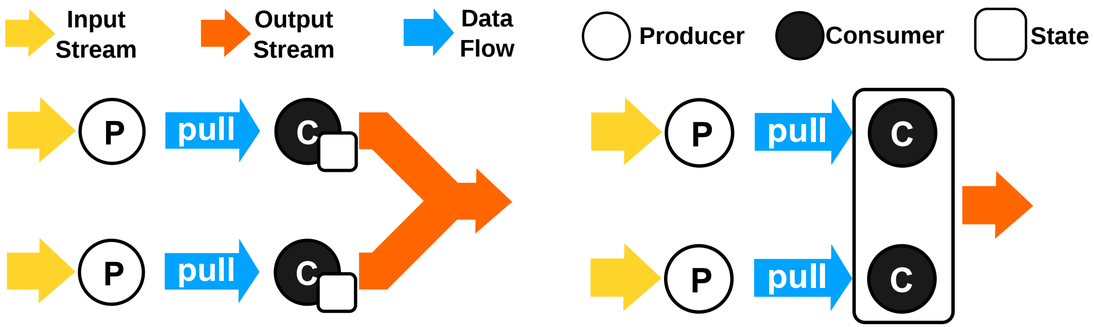
Proposed changes for *scale-up* design

Compute and merge partial states without data shuffling

Enable efficient data passing and code invocation

Parallelization: GM or LM instead of Upfront Partitioning

Hardware-tailored Query Compilation for Stream Processing



Late Merge (LM)

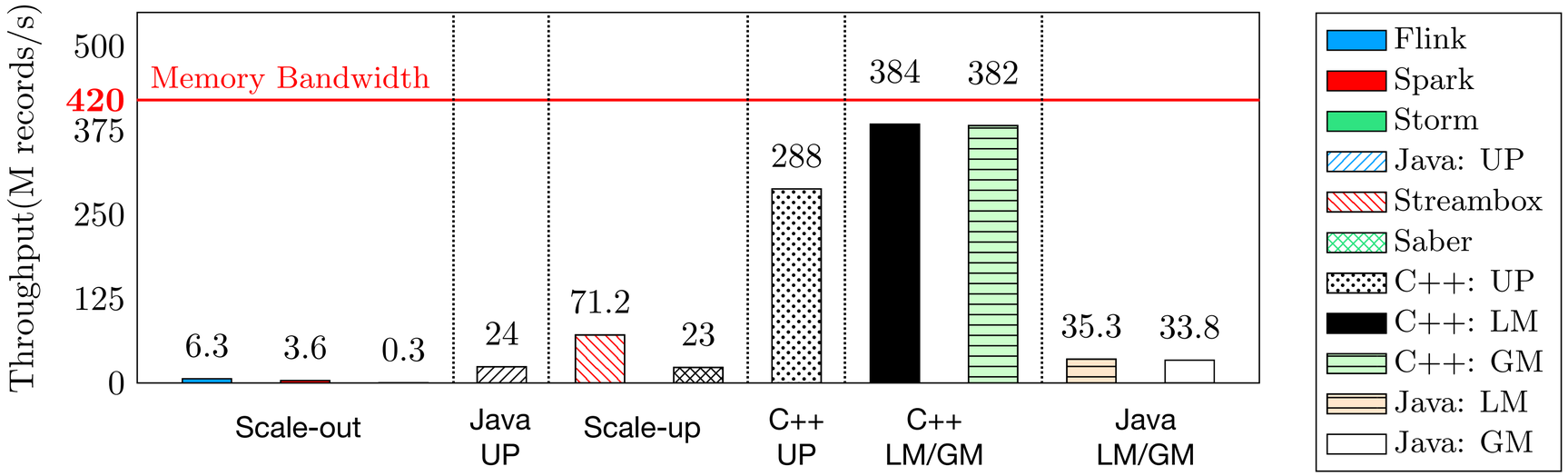
Global Merge (GM)

Operator Parallelization strategies

Compilation-based Query Execution

Overall efficient memory access patterns

Executing streaming queries on SPEs



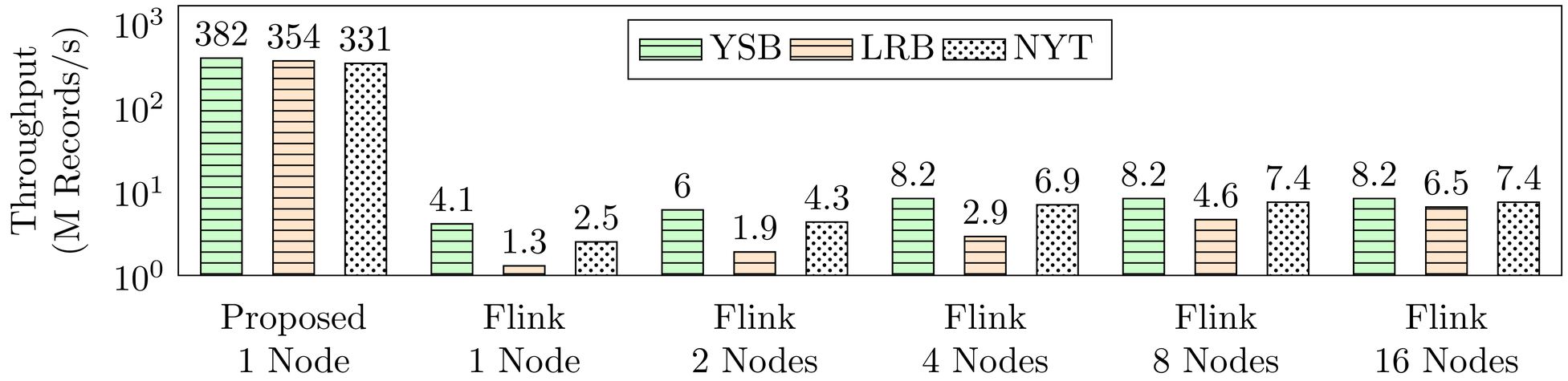
Parallelization: GM or LM instead of Upfront Partitioning

Hardware-tailored Query Compilation for Stream Processing

Avoid Managed Runtime (JVM)

C++ LM/GM achieve higher processing throughput

Scale-up is indeed better



Increasing node parallelism does not help

Summary

- SPEs are **CPU-Bound**: they **need design changes** to exploit modern hardware efficiently

- Propose **hardware-tailored query compilation** and **LM/GM operator parallelization to *scale-up*** stateful streaming queries

- **Two orders of magnitude** throughput improvement are possible

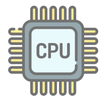
Agenda

SPEs are CPU-Bound

Understand Stream Processing Performance on Modern Hardware

E&A PVLDB 2019

1

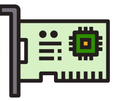


SPEs cannot fully use high-speed networks to scale-out

Efficient Scale-out Processing with High-speed Networks

SIGMOD 2022

2



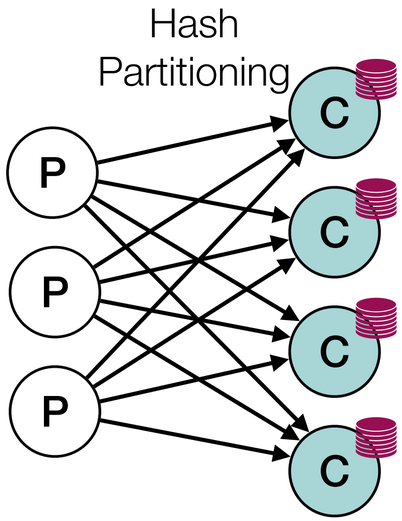
Large state is a bottleneck for on-the-fly query reconfiguration

Reliable Management of Very Large Operator State

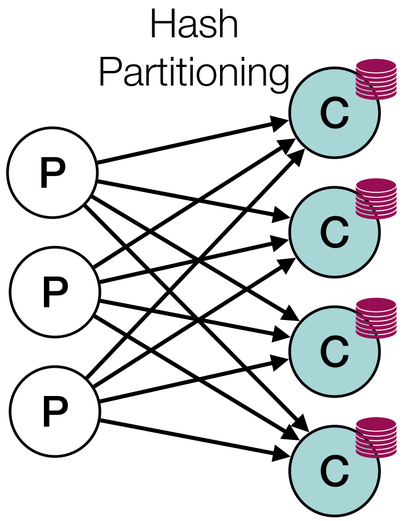
SIGMOD 2020

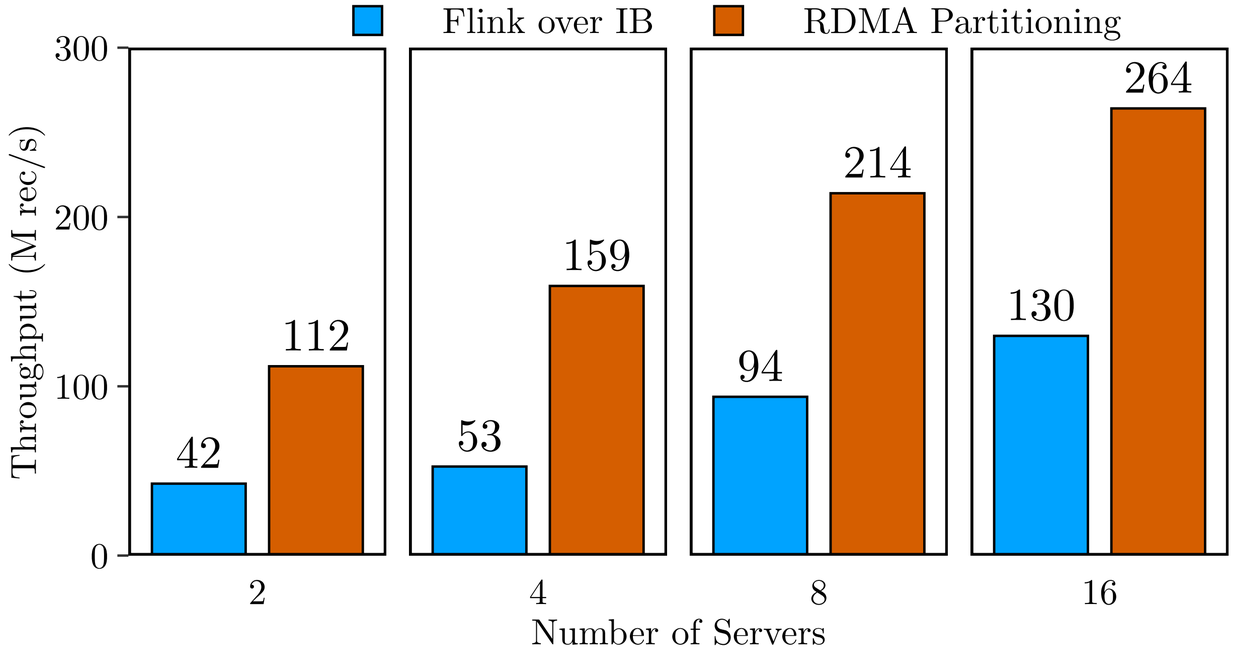
3

SPEs with high-speed network



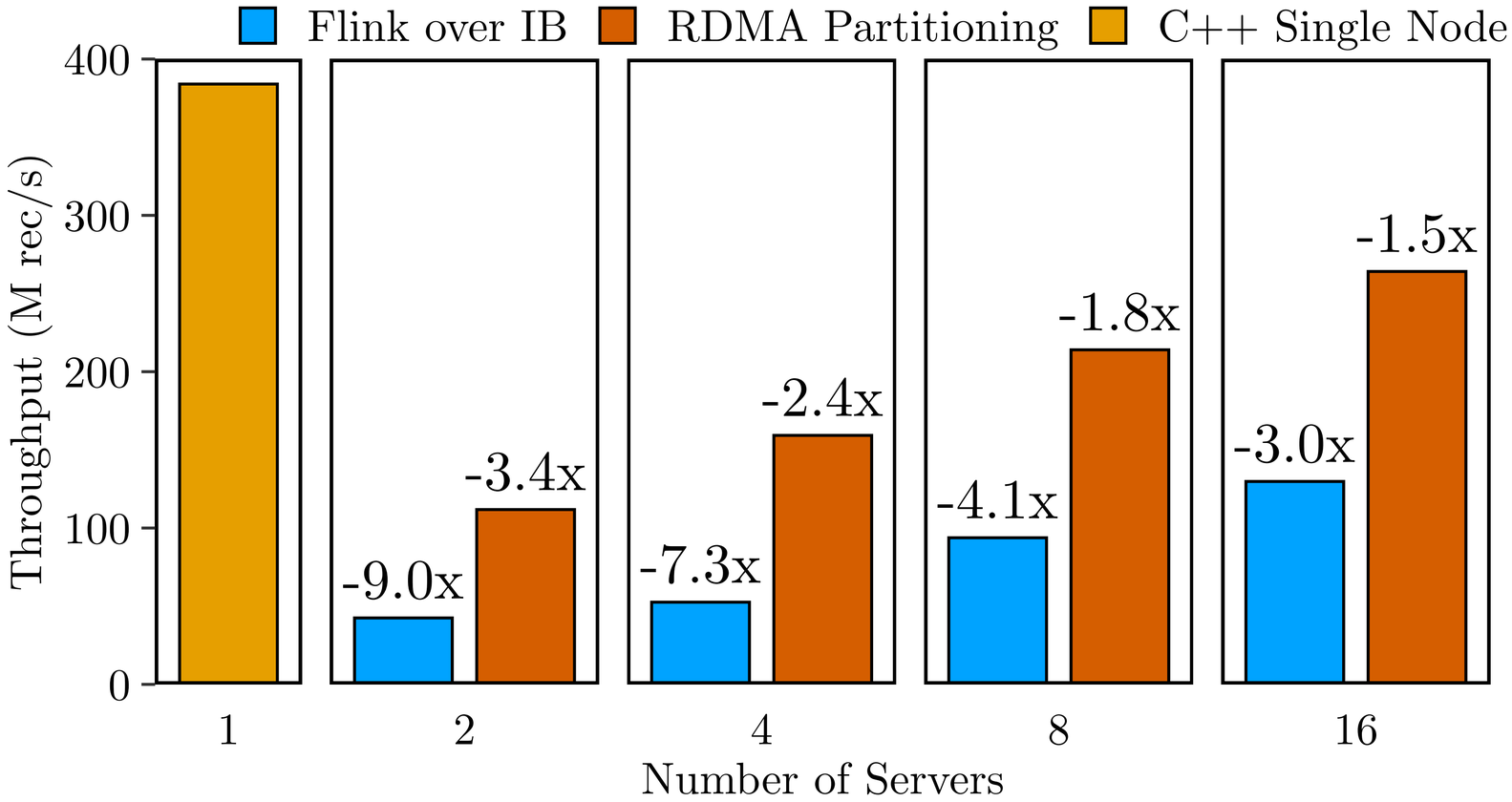
SPEs with high-speed network





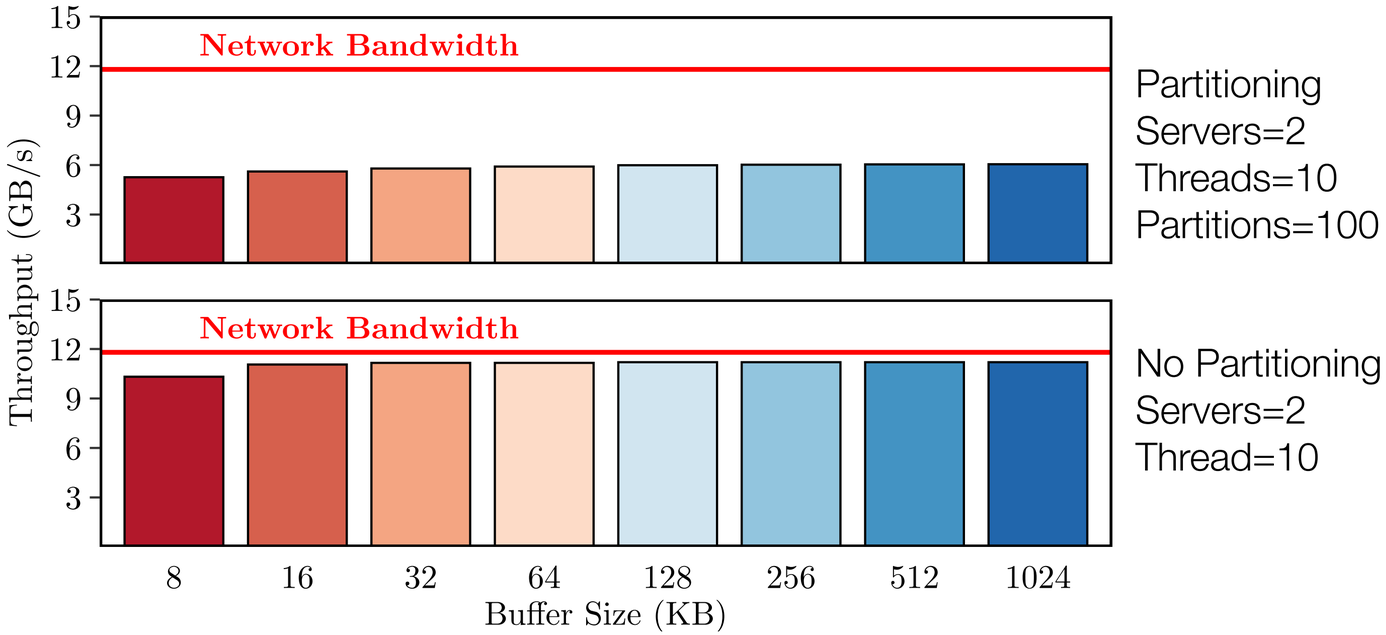
Intel Xeon Gold 5115 @ 2.4 Ghz 10-cores
RAM: 96GB
RNIC: Mellanox Connect-X4 EDR 100Gbps

SPEs with high-speed network



Simply using a high-speed network is not enough

Finding the bottleneck

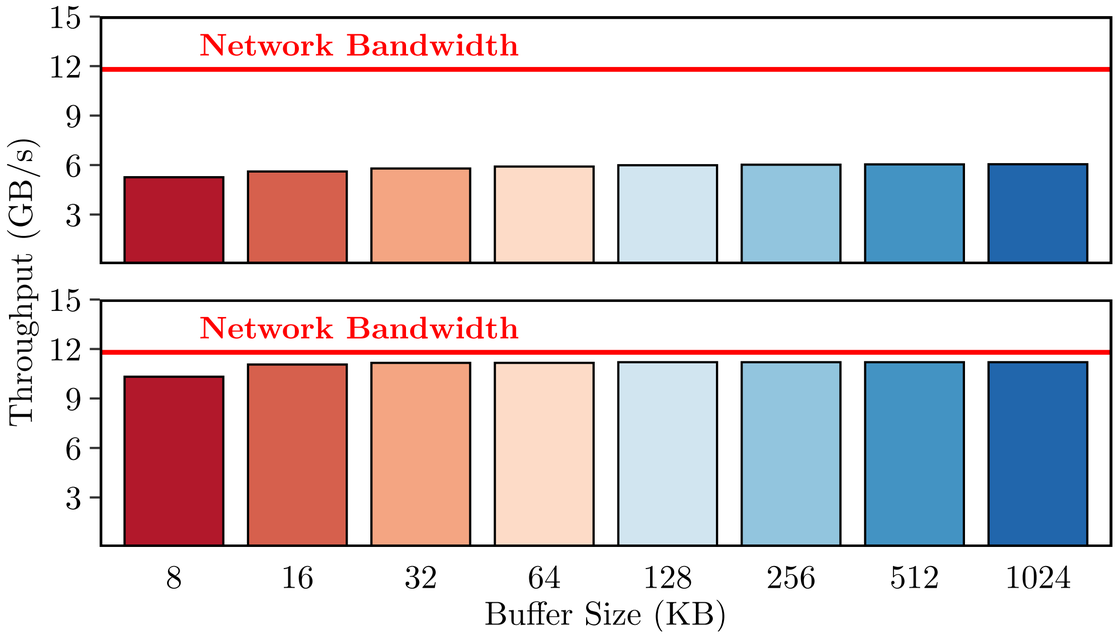


Partitioning
Servers=2
Threads=10
Partitions=100

No Partitioning
Servers=2
Thread=10

Data partitioning is a bottleneck also on two nodes

Finding the bottleneck



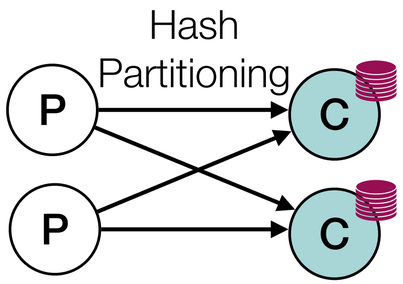
Partitioning
Servers=2
Threads=10
Partitions=100

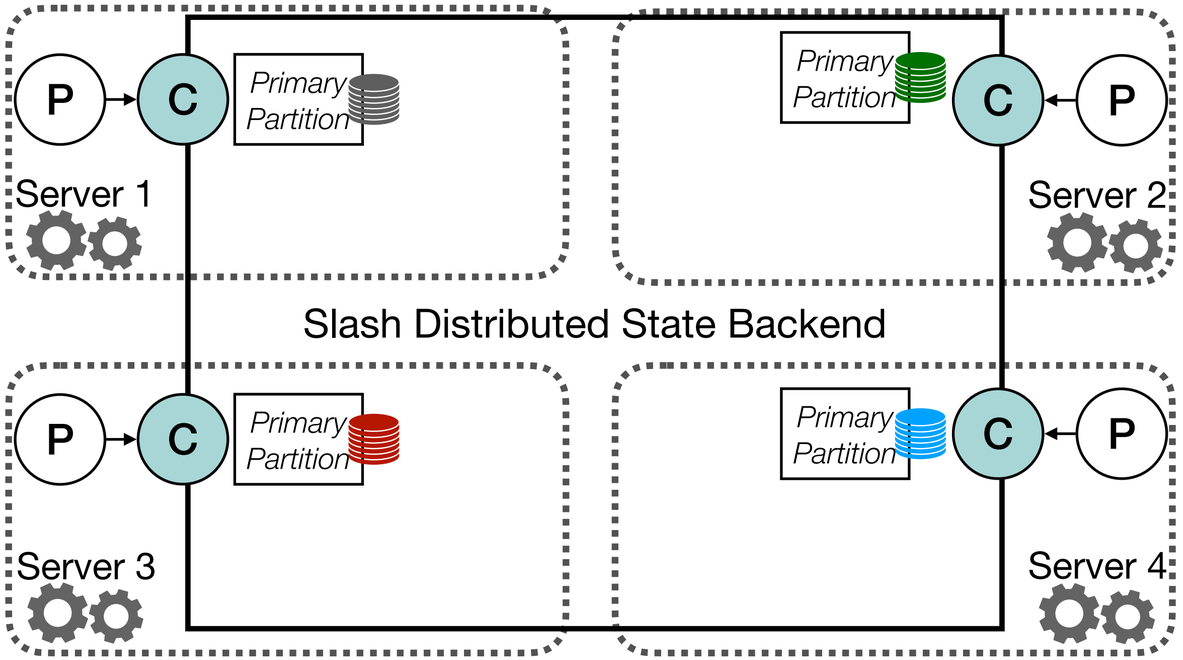
No Partitioning
Servers=2
Thread=10

Data partitioning is a bottleneck also on two nodes

Late Merge and Global Merge using distributed memory with RDMA

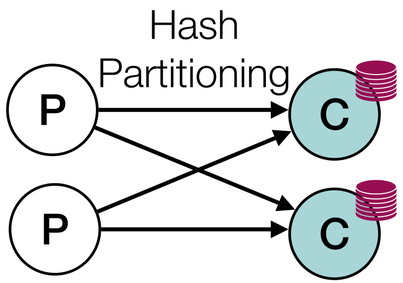
Slash: network-conscious SPE

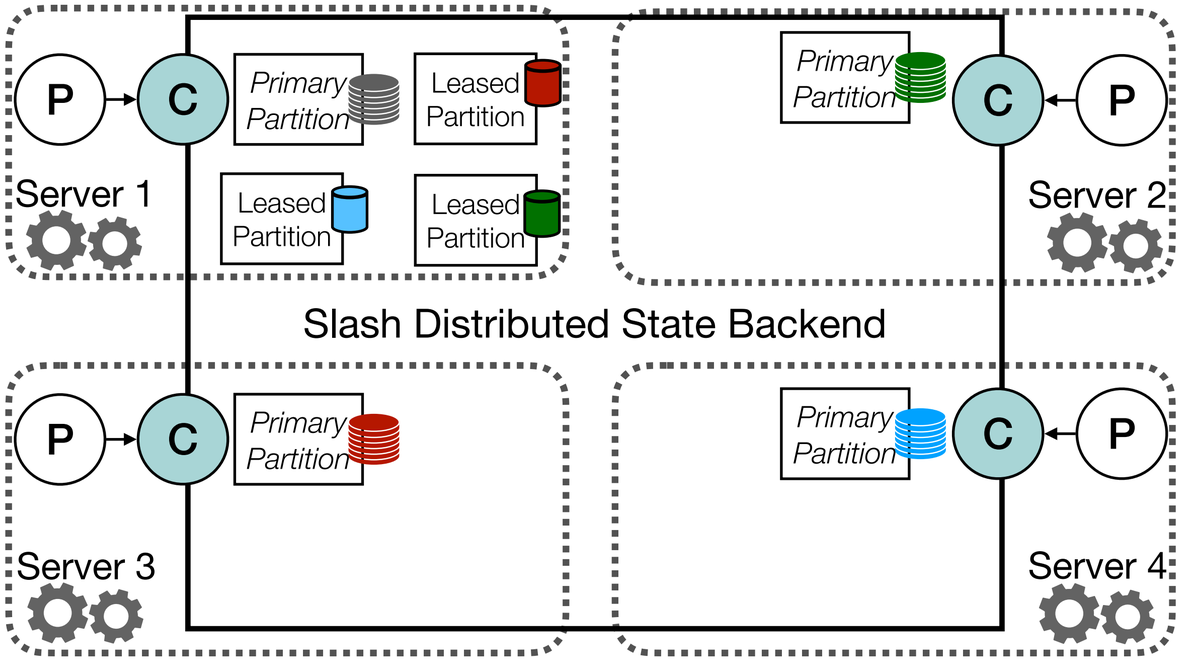




Primary Partitions: disjoint shards of operator state

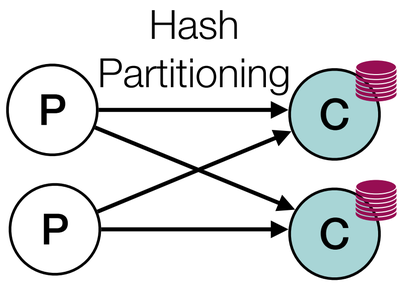
Slash: network-conscious SPE

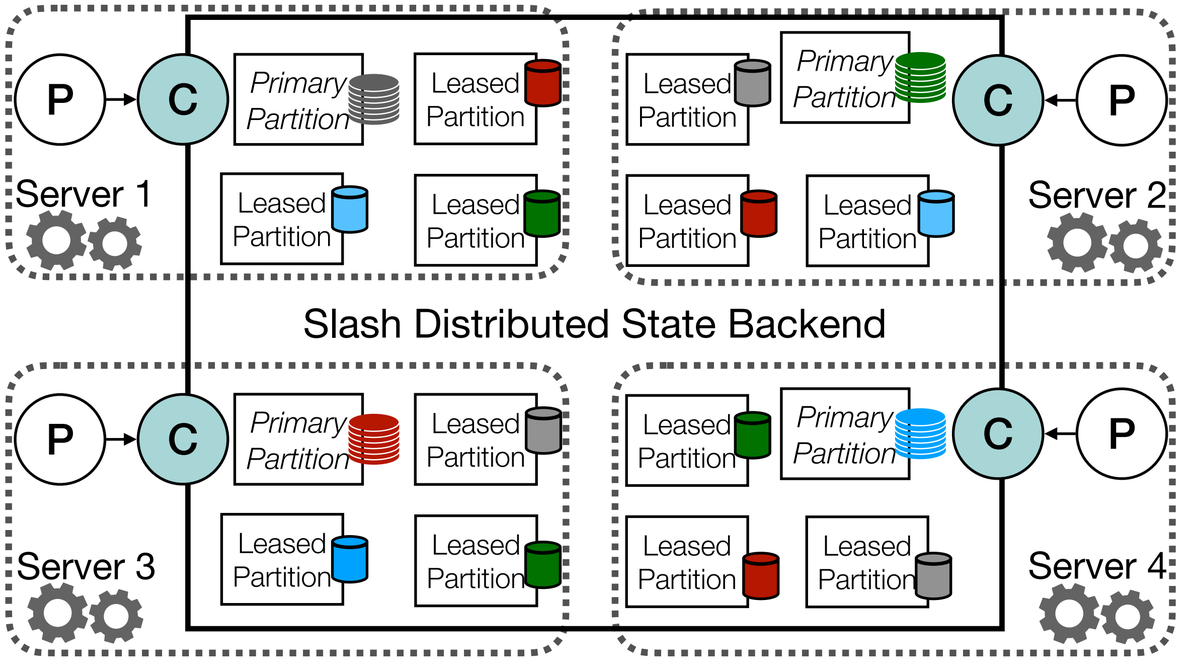




Primary Partitions: disjoint shards of operator state

Slash: network-conscious SPE

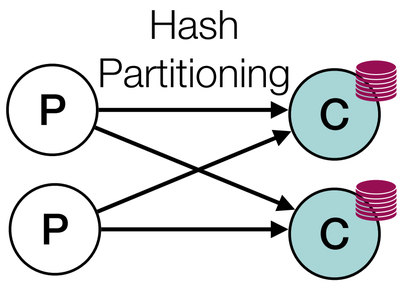


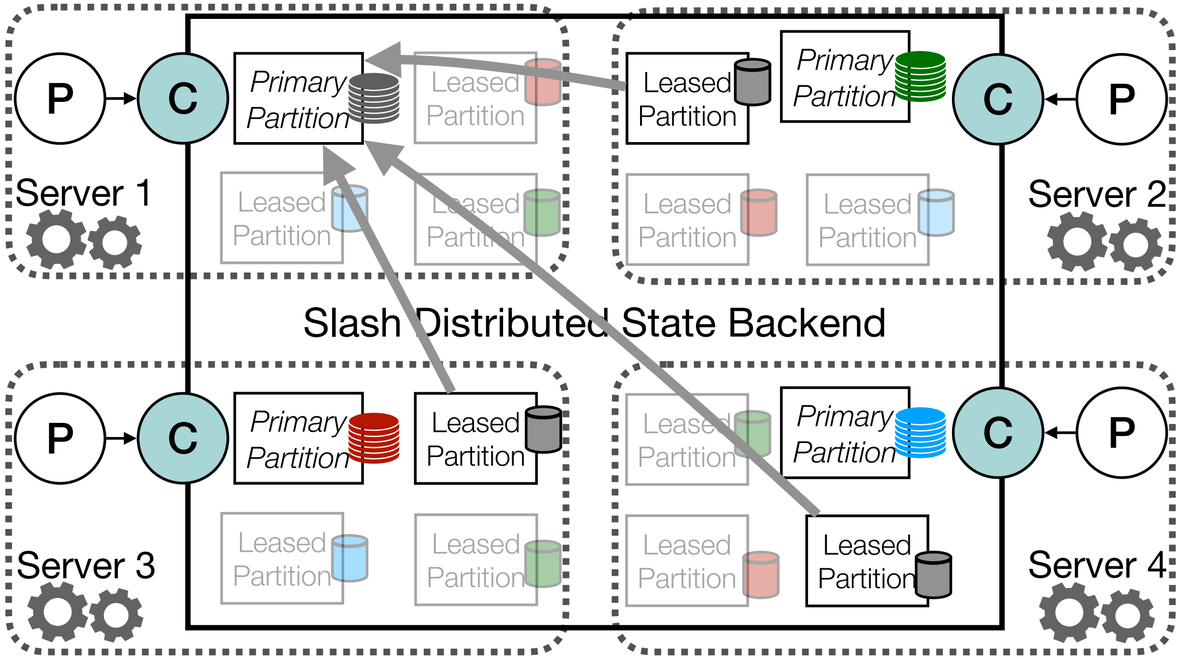


Primary Partitions: disjoint shards of operator state

Replace partitioning with eager computation of partial states followed by lazy merge

Slash: network-conscious SPE



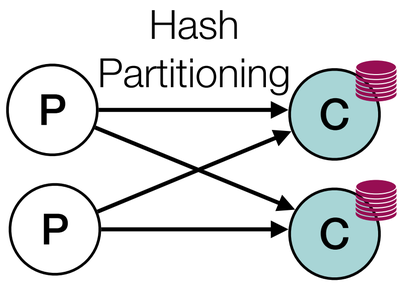


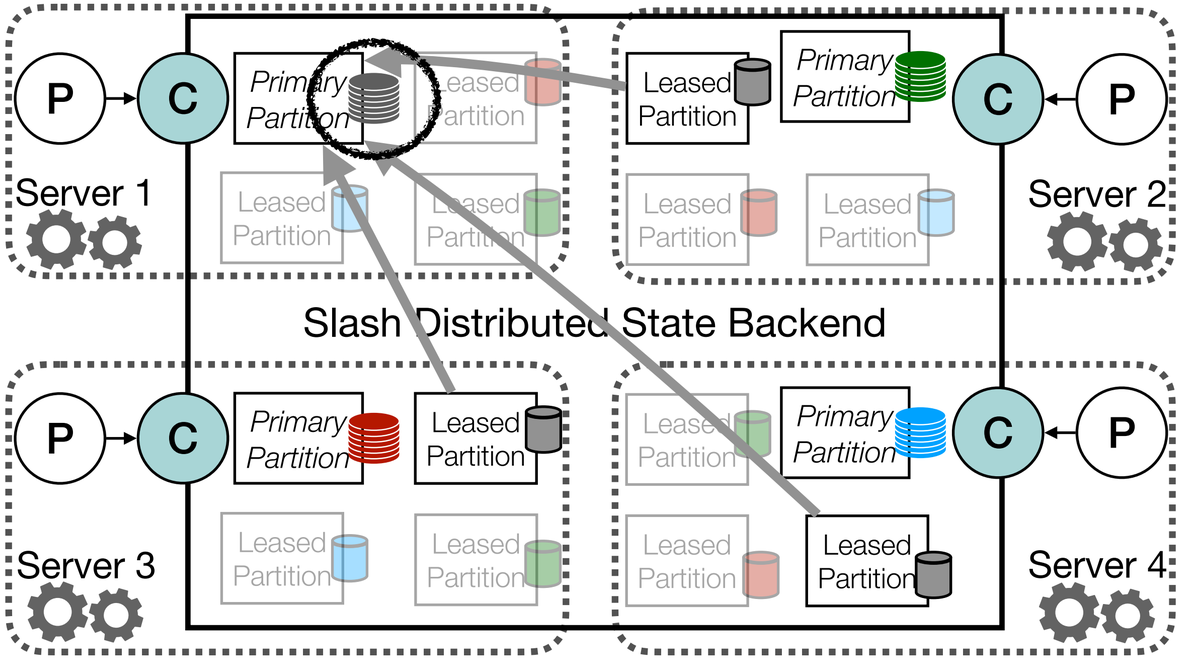
Primary Partitions: disjoint shards of operator state

Epoch-based synchro-nisation: to merge leased and primary partitions

Replace partitioning with eager computation of partial states followed by lazy merge

Slash: network-conscious SPE





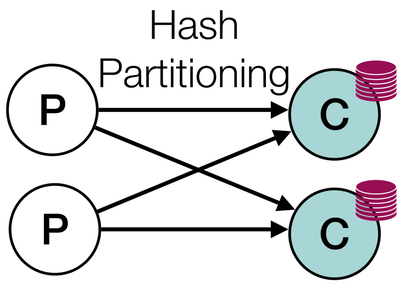
Primary Partitions: disjoint shards of operator state

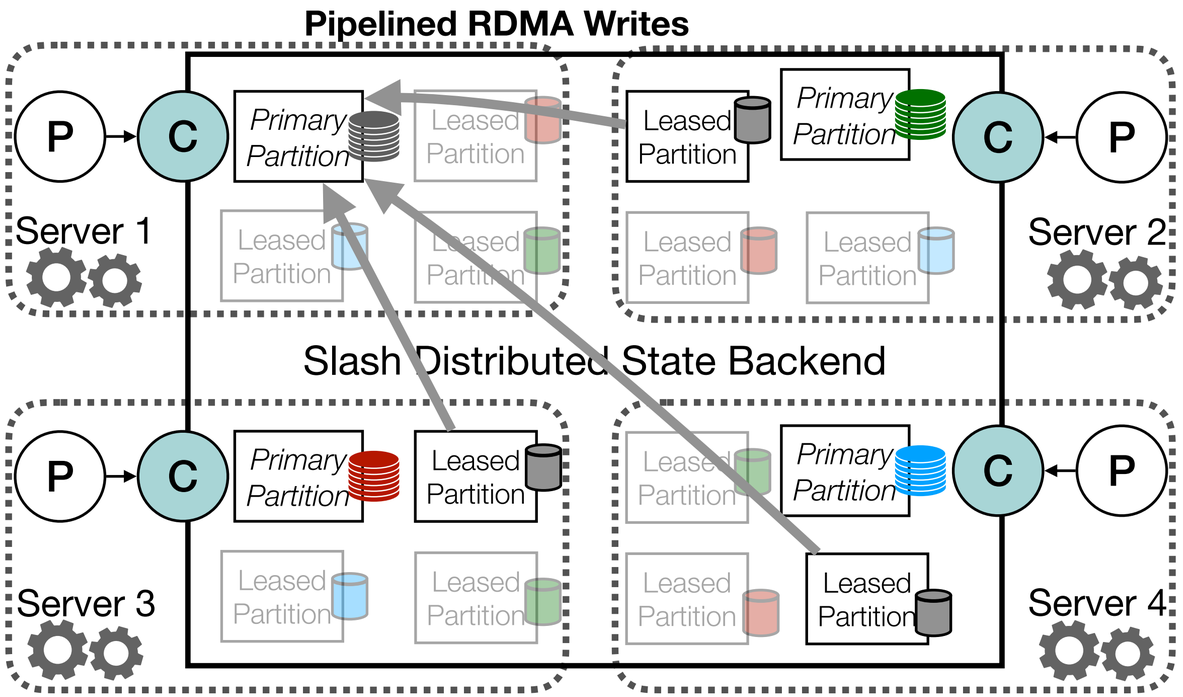
Epoch-based synchro-nisation: to merge leased and primary partitions

Conflict-free Replicated Data Types: to solve merge conflicts

Replace partitioning with eager computation of partial states followed by lazy merge

Slash: network-conscious SPE





Primary Partitions: disjoint shards of operator state

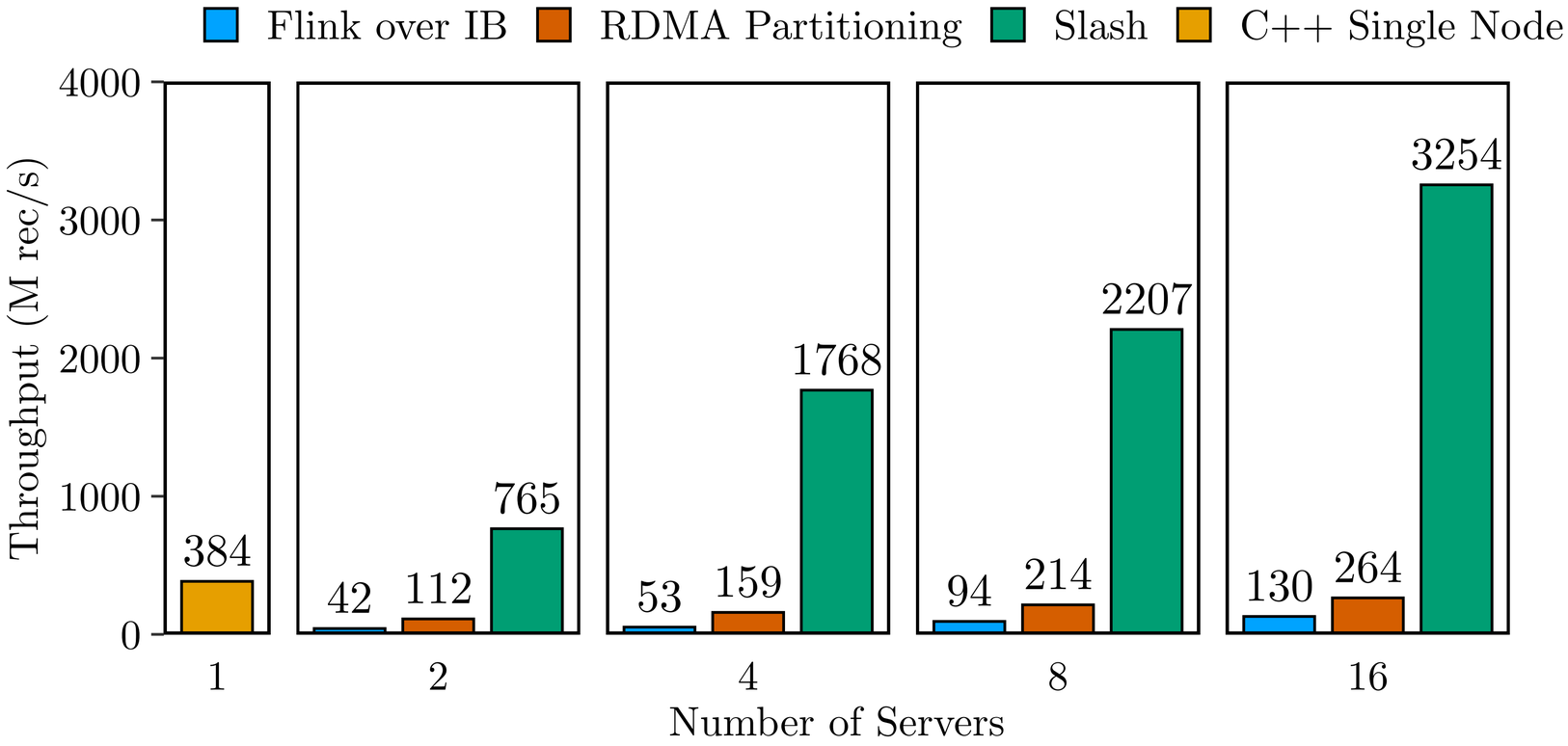
Epoch-based synchro-nisation: to merge leased and primary partitions

Conflict-free Replicated Data Types: to solve merge conflicts

Pipelined RDMA Writes: to transfer state chunks asynchronously

Replace partitioning with eager computation of partial states followed by lazy merge

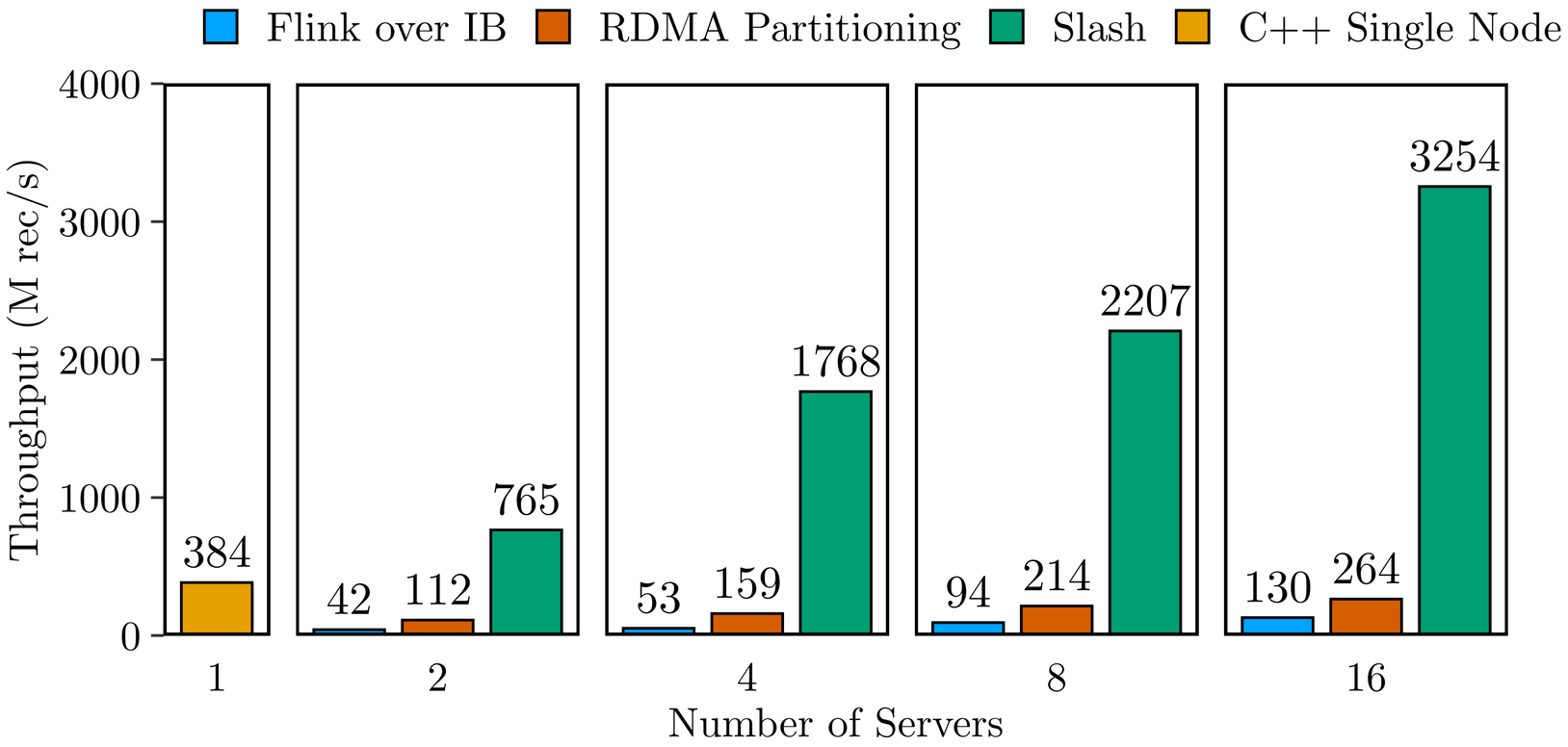
Performance of Slash



12x throughput improvement using 16 nodes

16-node Slash is 8x faster than optimised single node

Performance of Slash



RDMA baseline limited by partitioning speed (CPU-Bound)

Slash is limited by memory speed

Summary

- SPE design to **accelerate streaming workloads using RDMA** at rack-scale
- **No free lunch**: SPEs cannot efficiently scale-out using high-speed networks out-of-the-box
- Achieve **12x** throughput improvement over strongest baseline
- **Slash is memory-bound**; **baseline is bound by partitioning speed**

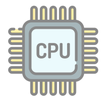
Hardware-conscious techniques for SPEs

SPEs are CPU-Bound

Understand Stream Processing Performance on Modern Hardware

E&A PVLDB 2019

1

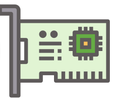


SPEs cannot fully use high-speed networks to scale-out

Efficient Scale-out Processing with High-speed Networks

SIGMOD 2022

2



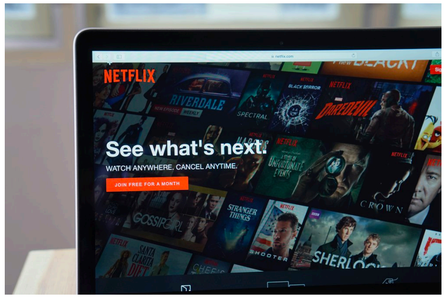
Large state is a bottleneck for on-the-fly query reconfiguration

Reliable Management of Very Large Operator State

SIGMOD 2020

3

Use case for large state

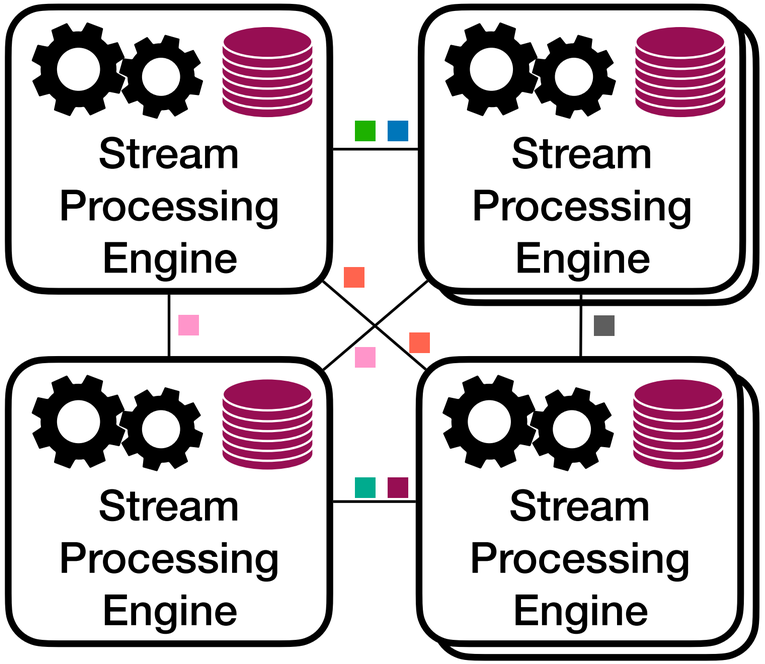


billions of events
high cardinality

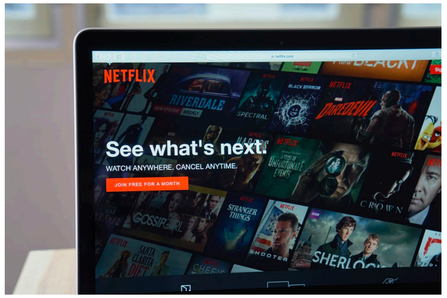
Credit Card Fraud Detection

data rate 1-10 **GB/s**

state size 1-10 **TB**

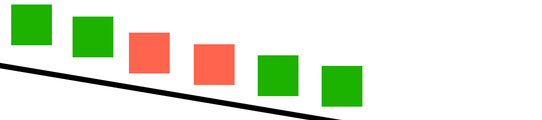


Anything that can go wrong will go wrong

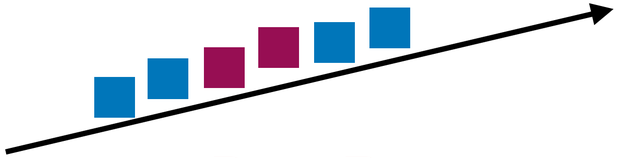


billions of events
high cardinality

Credit Card Fraud Detection

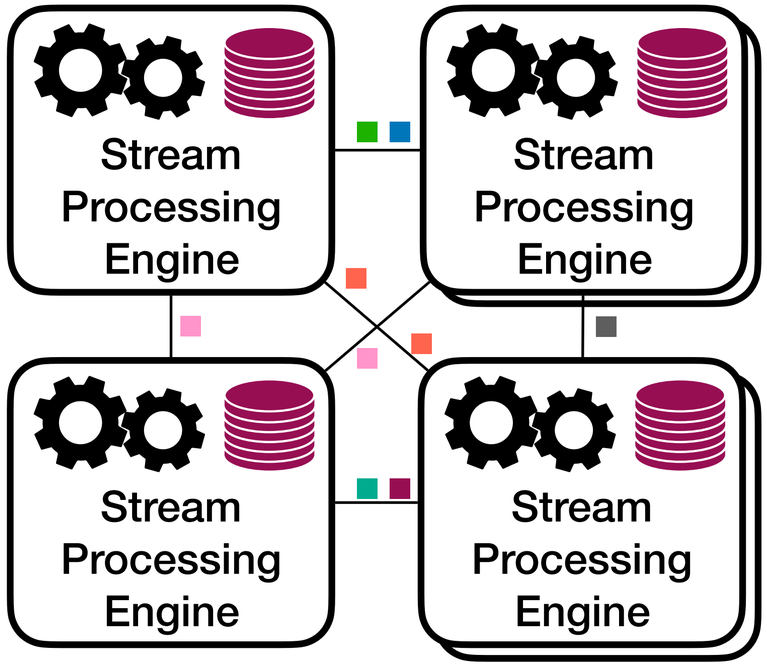


data rate 1-10 **GB/s**



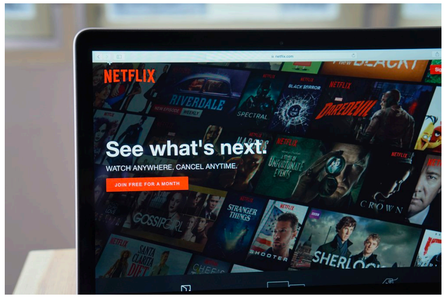
Data Rate Fluctuations

state size 1-10 **TB**



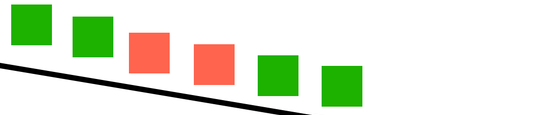
Failures

Anything that can go wrong will go wrong

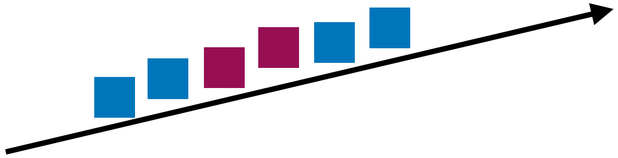


billions of events
high cardinality

Credit Card Fraud Detection

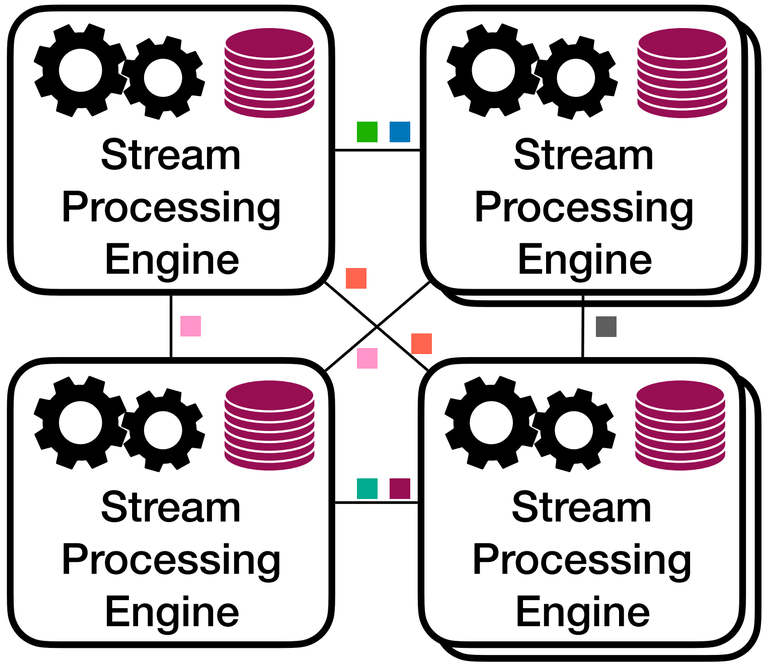


data rate 1-10 **GB/s**



Data Rate Fluctuations

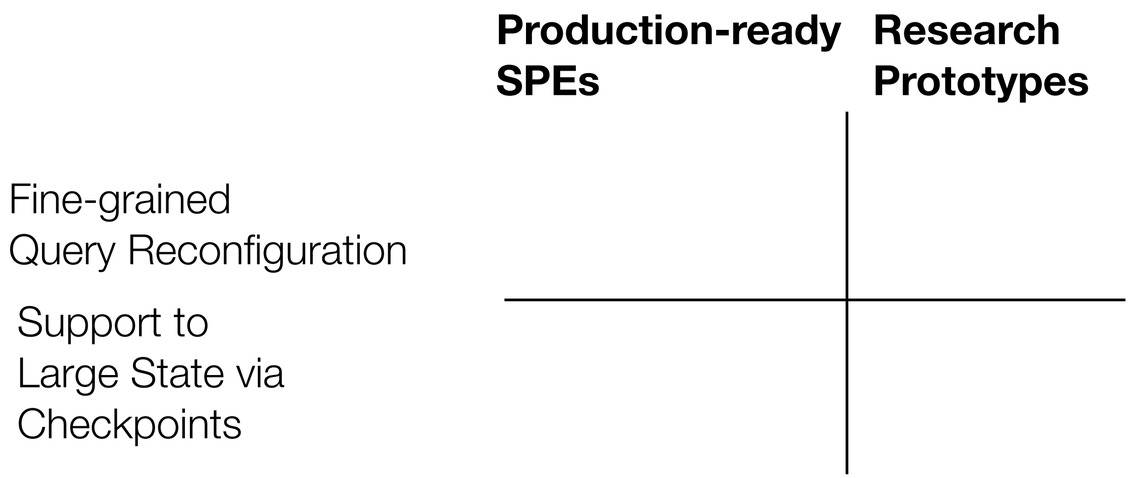
state size 1-10 **TB**



Failures

Slow query reconfiguration leads to high latency for query processing

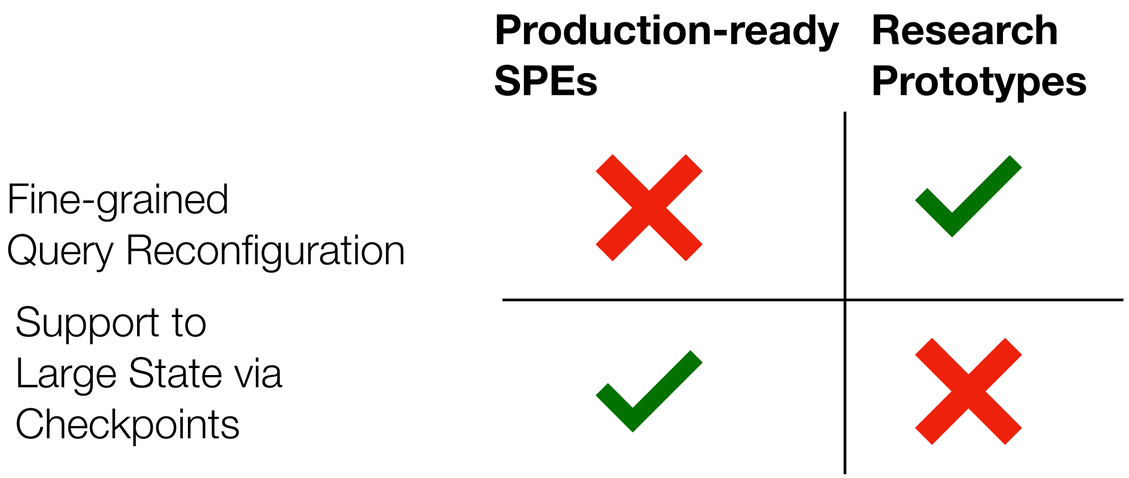
Benchmarking reconfiguration with large state



Benchmarking reconfiguration with large state

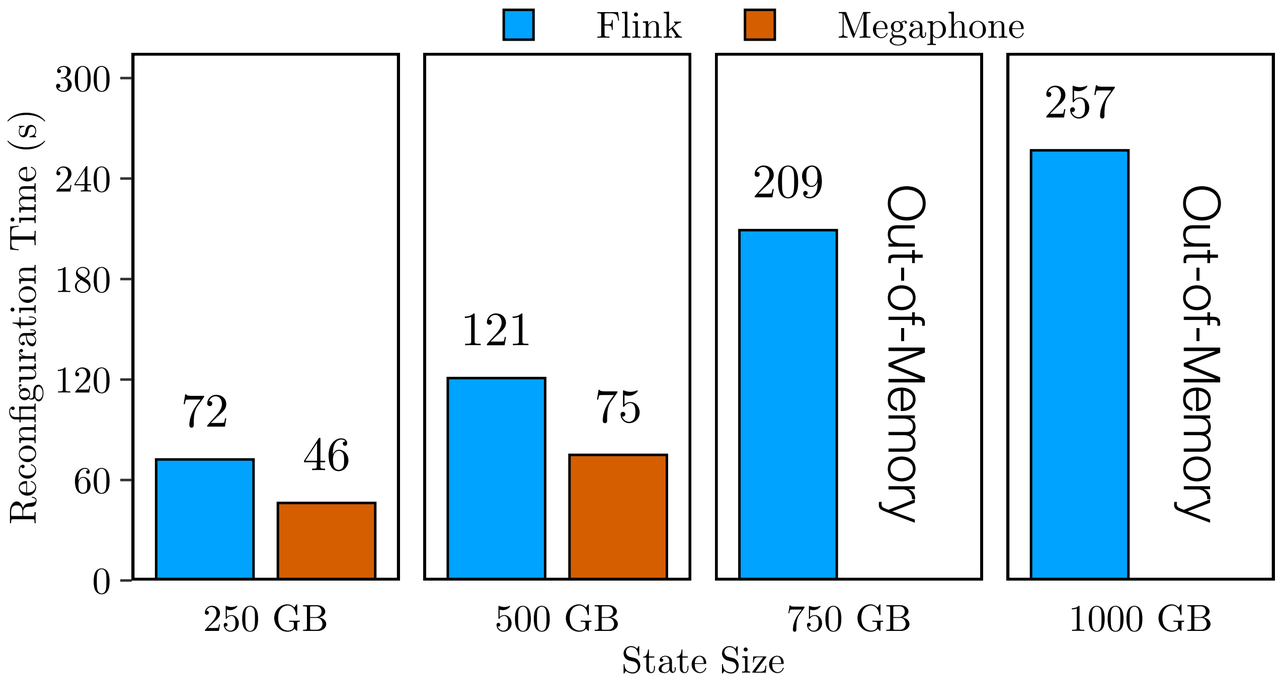
	Production-ready SPEs	Research Prototypes
Fine-grained Query Reconfiguration	✗	✓
Support to Large State via Checkpoints		

Benchmarking reconfiguration with large state



Benchmarking reconfiguration with large state

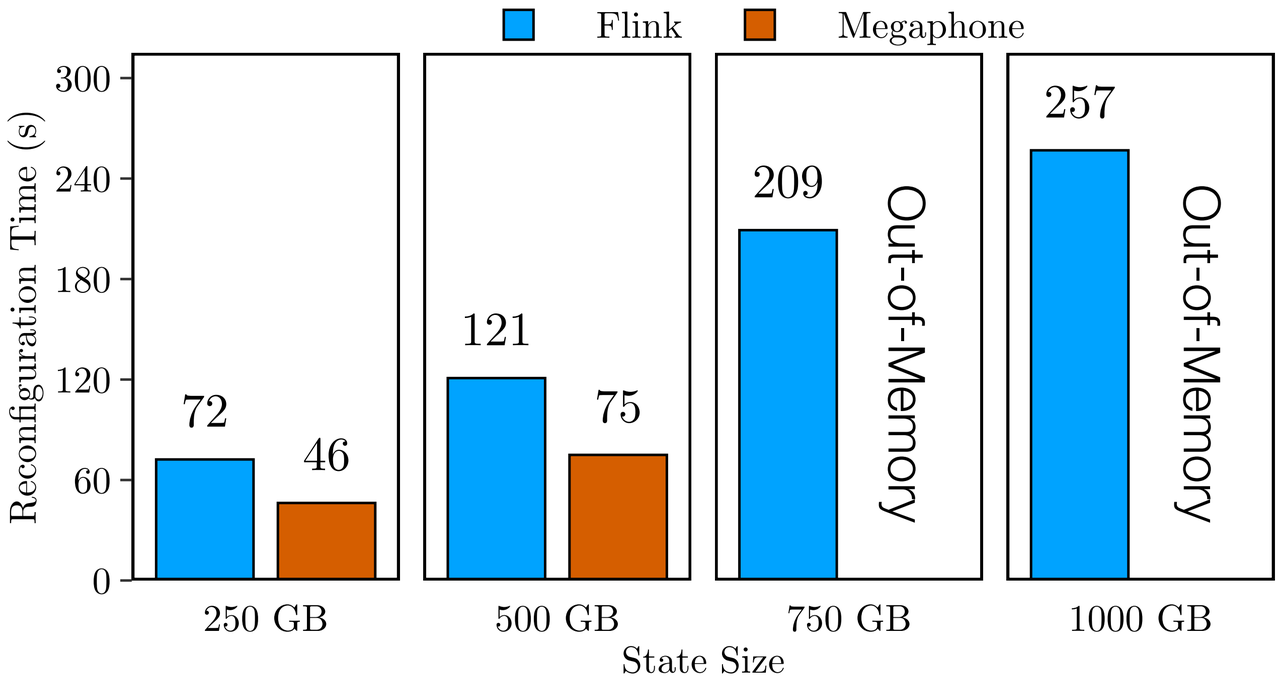
	Production-ready SPEs	Research Prototypes
Fine-grained Query Reconfiguration	✗	✓
Support to Large State via Checkpoints	✓	✗



8+1 n1-standard-64 VMs on GCP
16 vCPUs (Intel Xeon 8173M) + 64 GB RAM
750 GB NVMe SSD
2 Gbps per vCPU

Benchmarking reconfiguration with large state

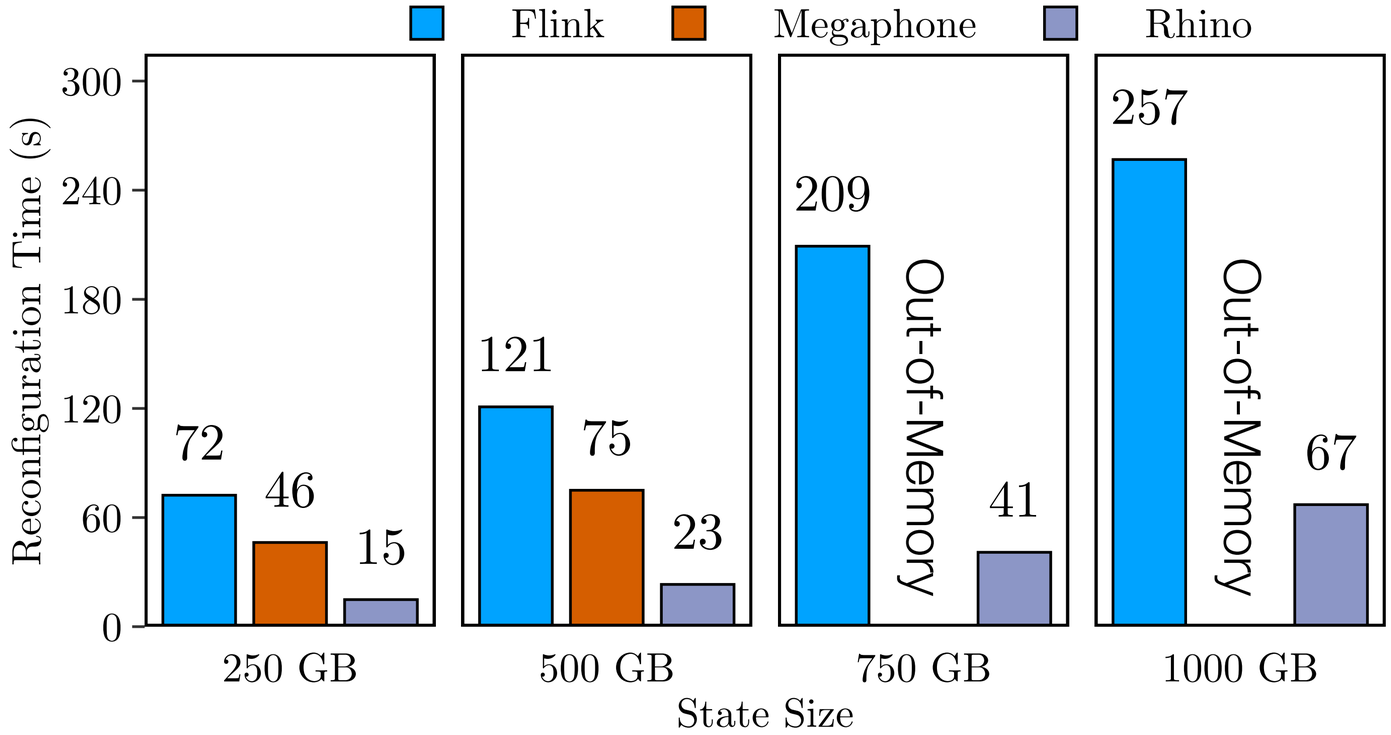
	Production-ready SPEs	Research Prototypes
Fine-grained Query Reconfiguration	✗	✓
Support to Large State via Checkpoints	✓	✗



We seek the best of both worlds: fine-grained query reconfiguration and support to large state

Our solution: Rhino

Handover Protocol to reconfigure running stateful query without halting it

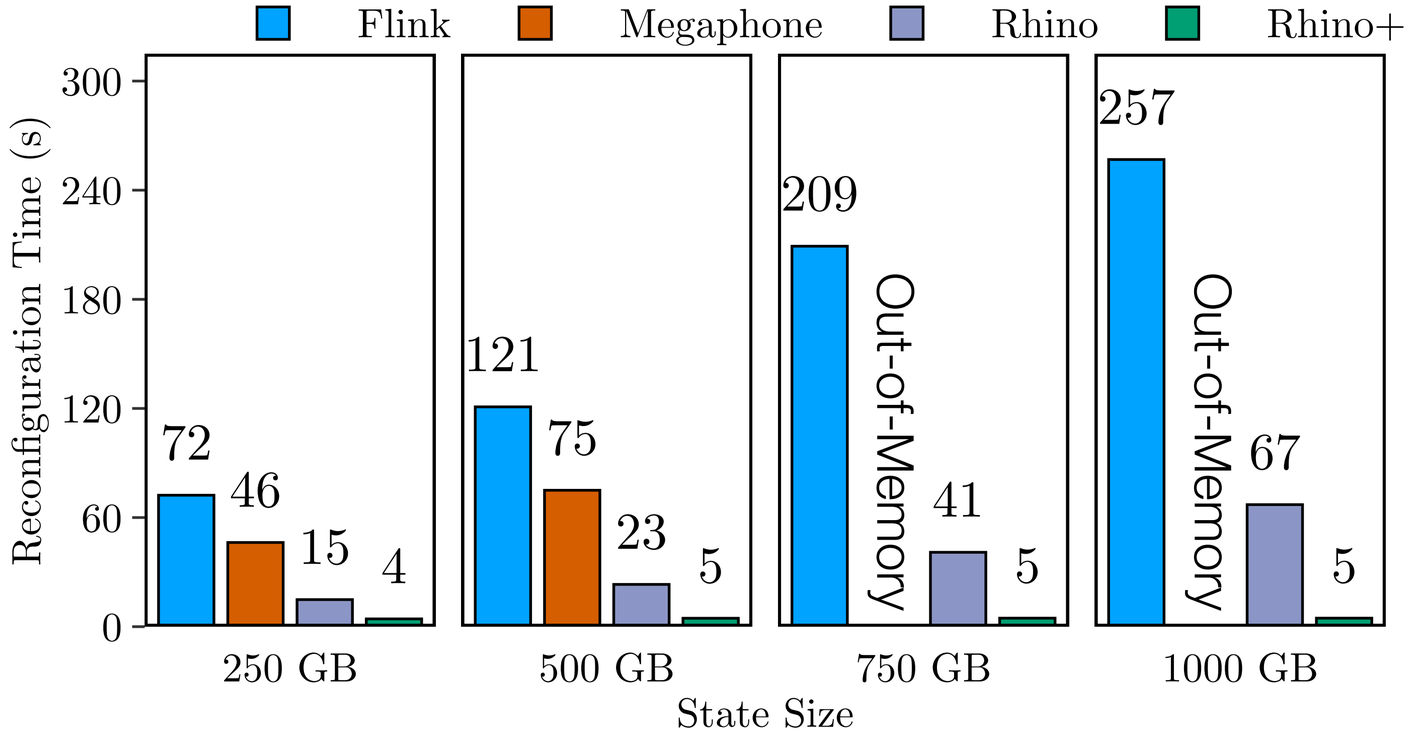


Rhino: decrement up to 3.8x with 1 TB state

Our solution: Rhino

Handover Protocol to reconfigure running stateful query without halting it

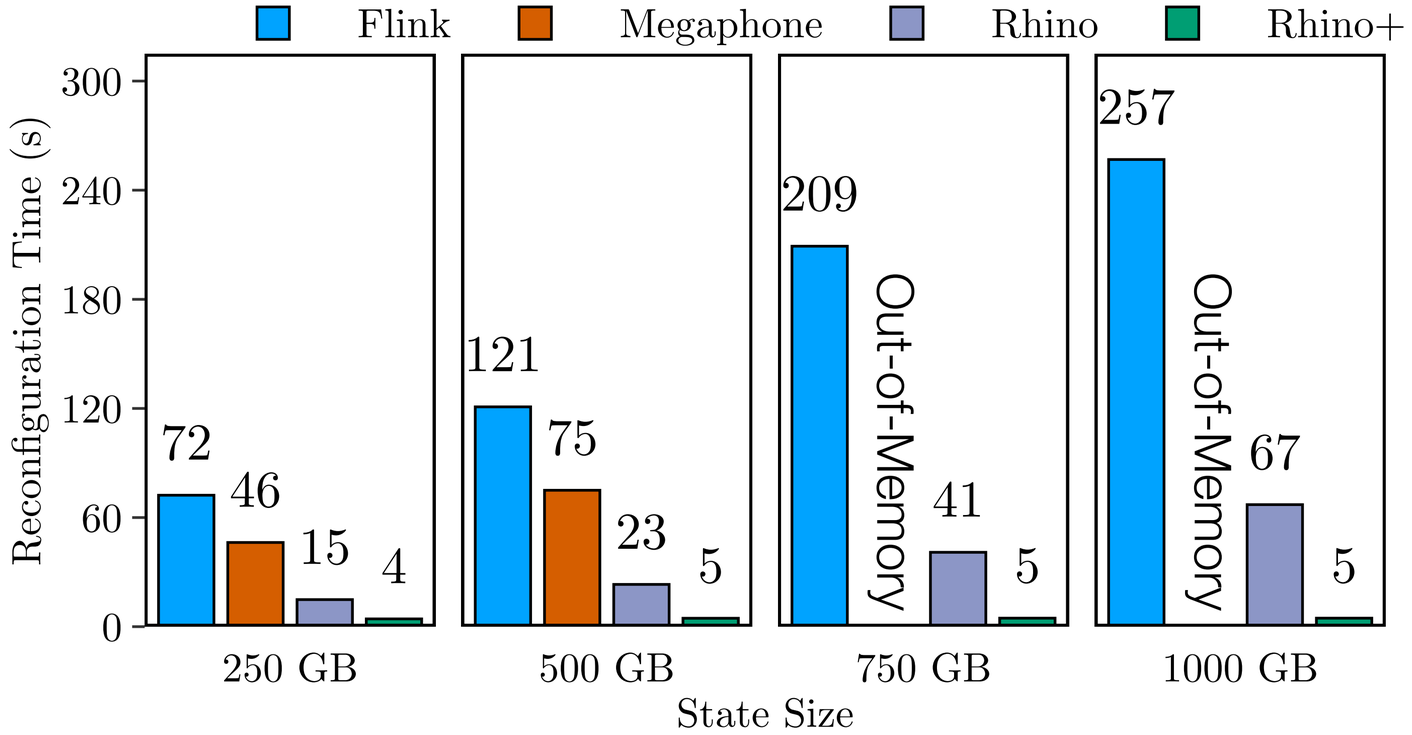
State Migration Protocol to proactively and incrementally replicate operator state among server



Our solution: Rhino

Handover Protocol to reconfigure running stateful query without halting it

State Migration Protocol to proactively and incrementally replicate operator state among server



Rhino+ reduces reconfiguration time by 3 orders of magnitude in the presence of TB-sized distributed operator state

Summary

- **Remove bottleneck** induced by large state migration upon query reconfiguration
- **Three orders of magnitude** query reconfiguration time **reduction**
- **Enable continuous SPE operations** by supporting *fault-tolerance, resource elasticity, and runtime reconfigurations* for running stateful queries

Conclusion

Hardware-oblivious SPE design results in performance issues

Conclusion

Hardware-oblivious SPE design results in performance issues

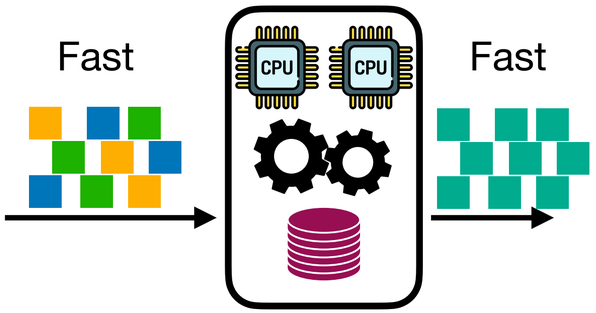
SPEs are CPU-Bound with high-speed networks

Conclusion

Hardware-oblivious SPE design results in performance issues

SPEs are CPU-Bound with high-speed networks

Query Compilation & Late/Global Merge

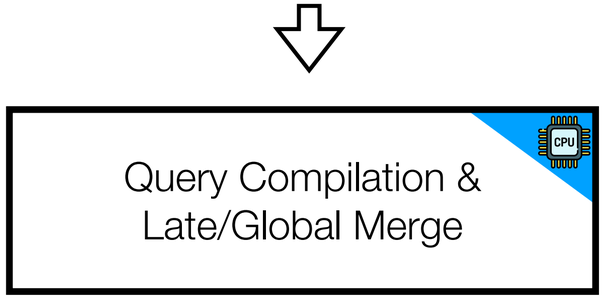


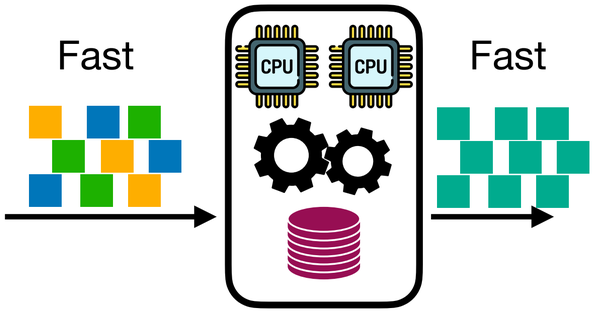
Conclusion

Hardware-oblivious SPE design results in performance issues

SPEs are CPU-Bound with high-speed networks

SPEs cannot fully use high-speed networks to scale-out



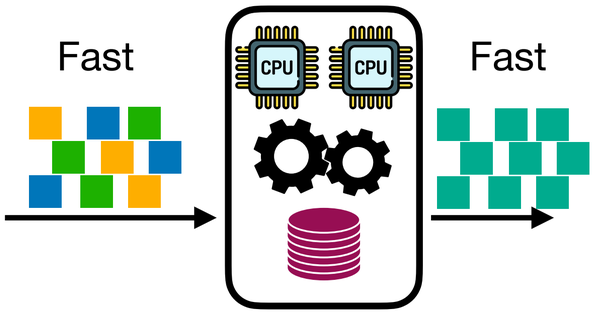


Conclusion

Hardware-oblivious SPE design results in performance issues

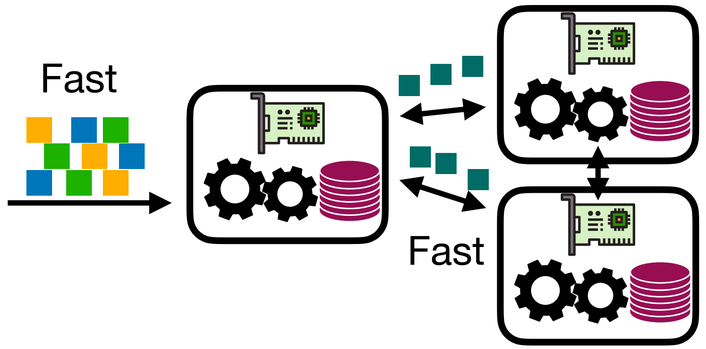
SPEs are CPU-Bound with high-speed networks

Query Compilation & Late/Global Merge



SPEs cannot fully use high-speed networks to scale-out

Partial State Computation & Lazy Merge using RDMA



Conclusion

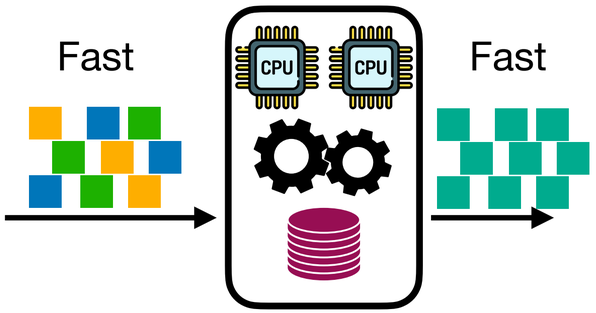
Hardware-oblivious SPE design results in performance issues

SPEs are CPU-Bound with high-speed networks

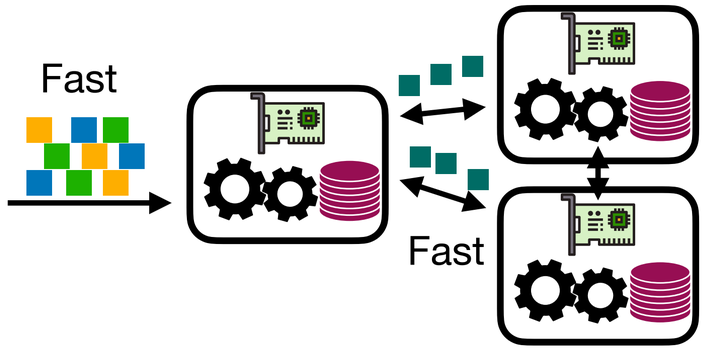
SPEs cannot fully use high-speed networks to scale-out

Large state is a bottleneck for on-the-fly query reconfiguration

Query Compilation & Late/Global Merge



Partial State Computation & Lazy Merge using RDMA

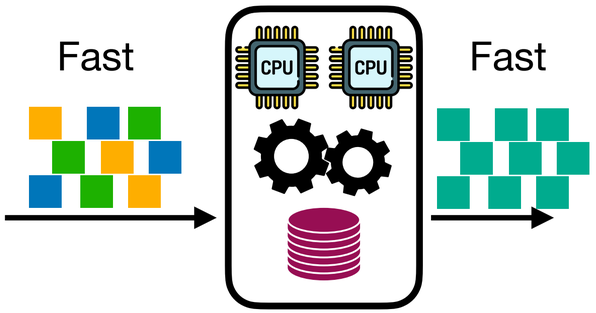


Conclusion

Hardware-oblivious SPE design results in performance issues

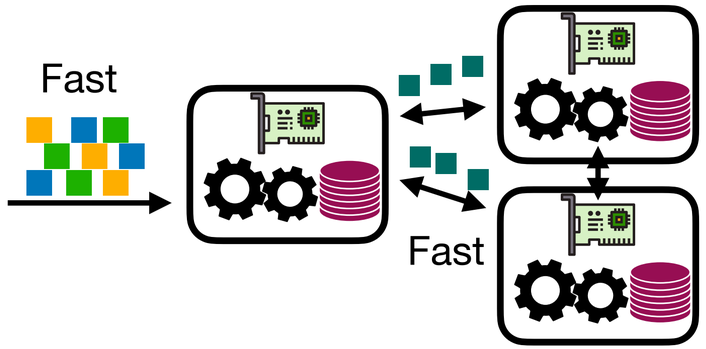
SPEs are CPU-Bound with high-speed networks

Query Compilation & Late/Global Merge



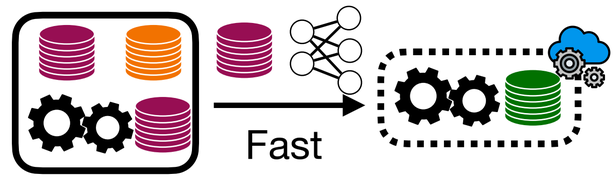
SPEs cannot fully use high-speed networks to scale-out

Partial State Computation & Lazy Merge using RDMA



Large state is a bottleneck for on-the-fly query reconfiguration

Fine-grained Query Reconfiguration & Proactive State Migration



Conclusion

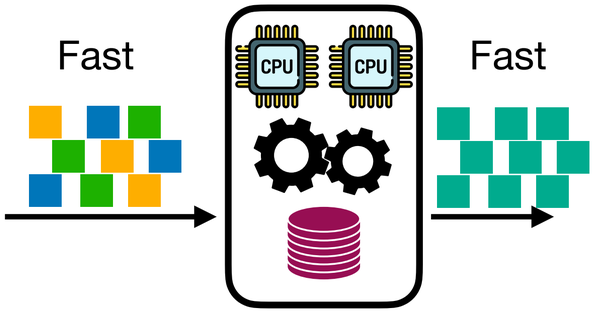
Hardware-conscious techniques enable efficient and reliable stateful query execution

~~**Hardware-oblivious SPE design results in performance issues**~~

SPEs are CPU-Bound with high-speed networks

⇩

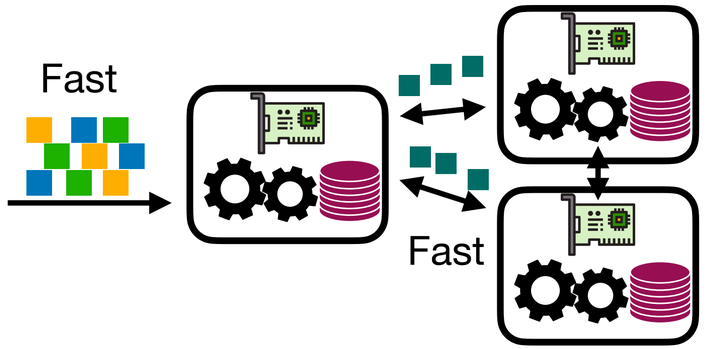
Query Compilation & Late/Global Merge



SPEs cannot fully use high-speed networks to scale-out

⇩

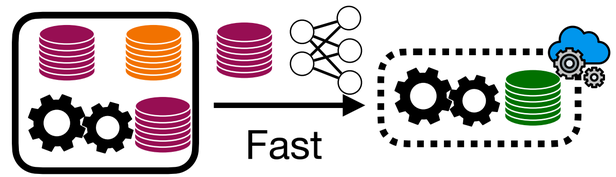
Partial State Computation & Lazy Merge using RDMA



Large state is a bottleneck for on-the-fly query reconfiguration

⇩

Fine-grained Query Reconfiguration & Proactive State Migration



Conclusion

Hardware-conscious techniques enable efficient and reliable stateful query execution

~~**Hardware-oblivious SPE design results in performance issues**~~

SPEs are CPU-Bound with high-speed networks

⇩

Query Compilation & Late/Global Merge

SPEs cannot fully use high-speed networks to scale-out

⇩

Partial State Computation & Lazy Merge using RDMA

Large state is a bottleneck for on-the-fly query reconfiguration

⇩

Fine-grained Query Reconfiguration & Proactive State Migration

NebulaStream

Conclusion

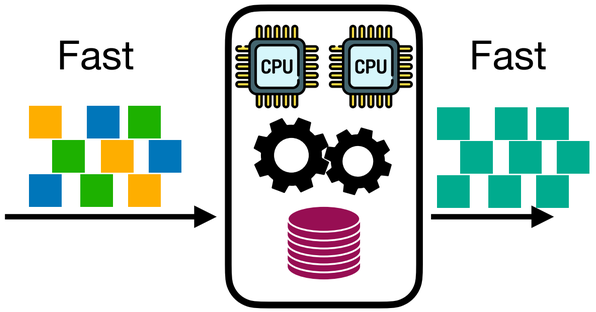
Hardware-conscious techniques enable efficient and reliable stateful query execution

~~**Hardware-oblivious SPE design results in performance issues**~~

SPEs are CPU-Bound with high-speed networks

⇩

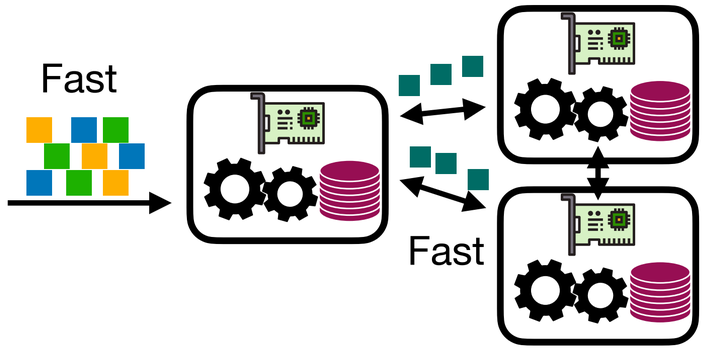
Query Compilation & Late/Global Merge



SPEs cannot fully use high-speed networks to scale-out

⇩

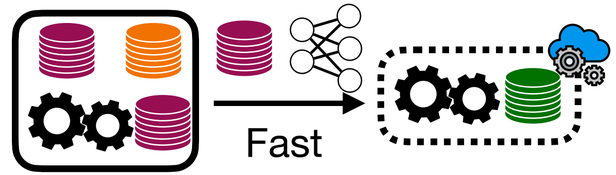
Partial State Computation & Lazy Merge using RDMA



Large state is a bottleneck for on-the-fly query reconfiguration

⇩

Fine-grained Query Reconfiguration & Proactive State Migration



Thank you!

Backup

Publications and contributions

- Efficient Scale-up Stateful Stream Processing @ PVLDB 2019
- Efficient Scale-out Stateful Stream Processing @ SIGMOD 2022
- Efficient State Management @ SIGMOD 2020
- Ph.D. Proposal @ VLDB Ph.D. Workshop 2017
- State Migration PoC @ BTW 2019
- NebulaStream Platform @ CIDR 2020 & VLIOT 2021

Ph.D. lessons learned

- Research-oriented coursework helps
  - I didn’t do that in my M.Sc., had to learn on the way at DIMA
- Idea -> Prototype -> Prove point -> Write paper sections -> Repeat
  - Quick validation, paper is written step-by-step, full system at the end
- Don’t ever use different plotting libraries
  - ..or you will have lots of fun by the time of your thesis submission/defense
- Check health of your experiment hardware

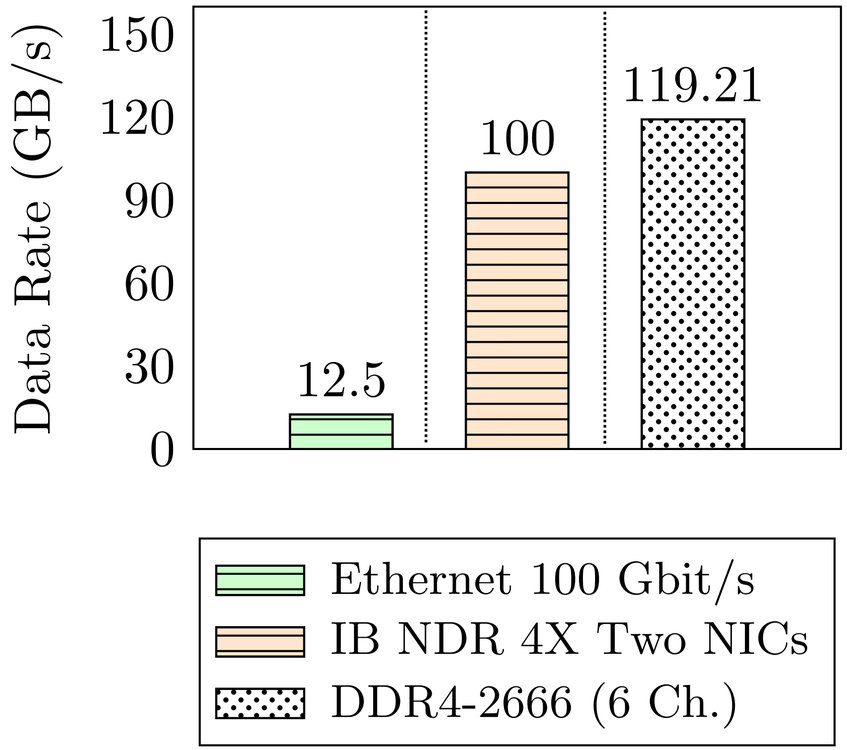
Research Outlook

- Internet-of-Things & ~~Stream Processing~~ Data Management
  - Distributed Query Execution, Optimizing Compiler, and State Management
  - Fault tolerance, Resource Scheduling/Optimization
- Disaggregated Resources in Datacenter
  - Implications on the design of data management systems
  - CXL and “Resource Blades”
- Do research closer to “real-world” application needs

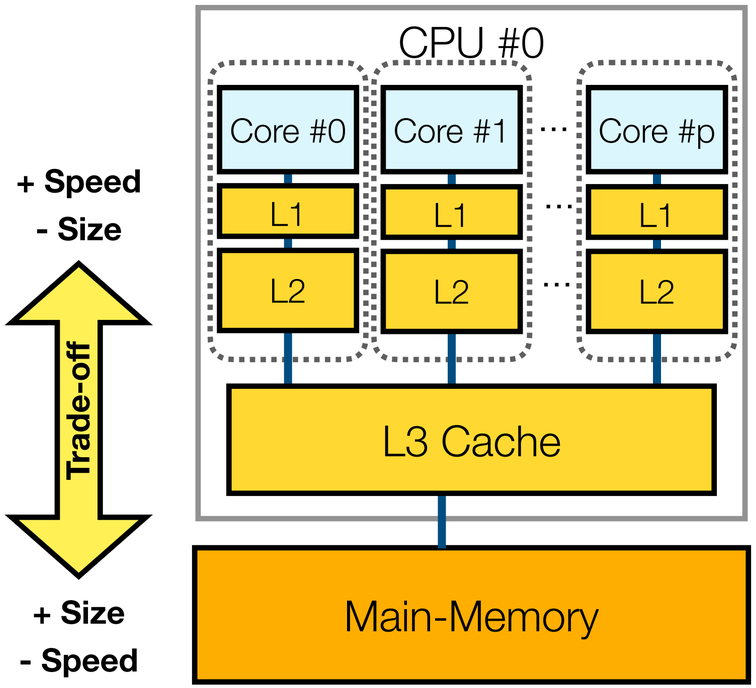
Backup

Understanding Stream Processing Performance

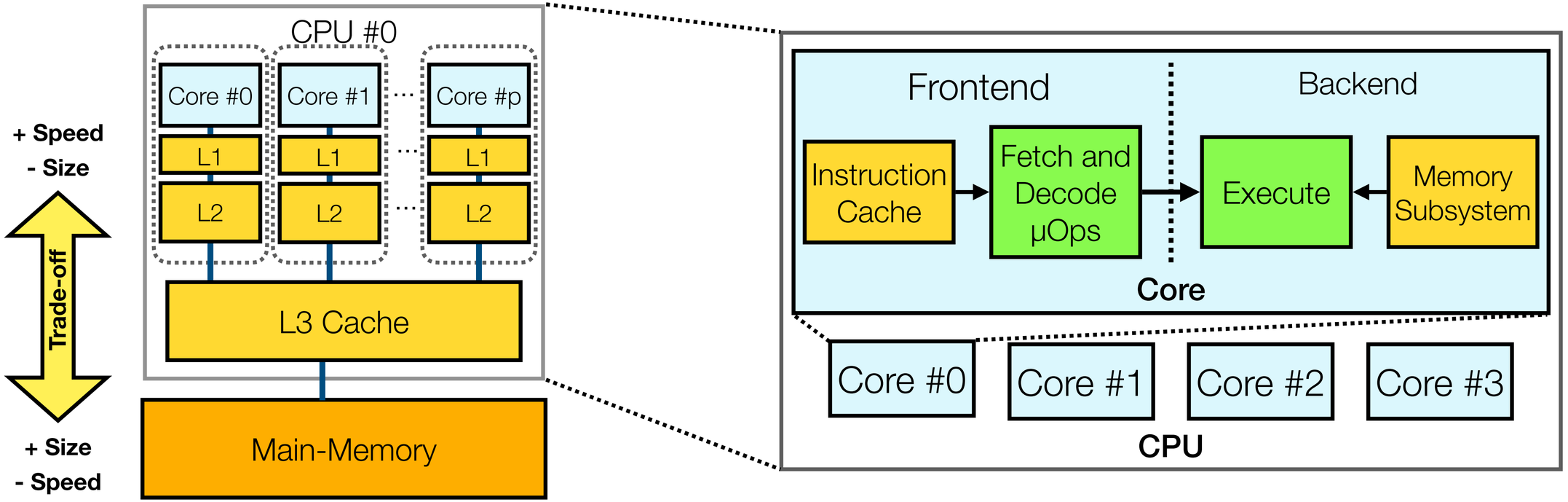
Today’s network speed



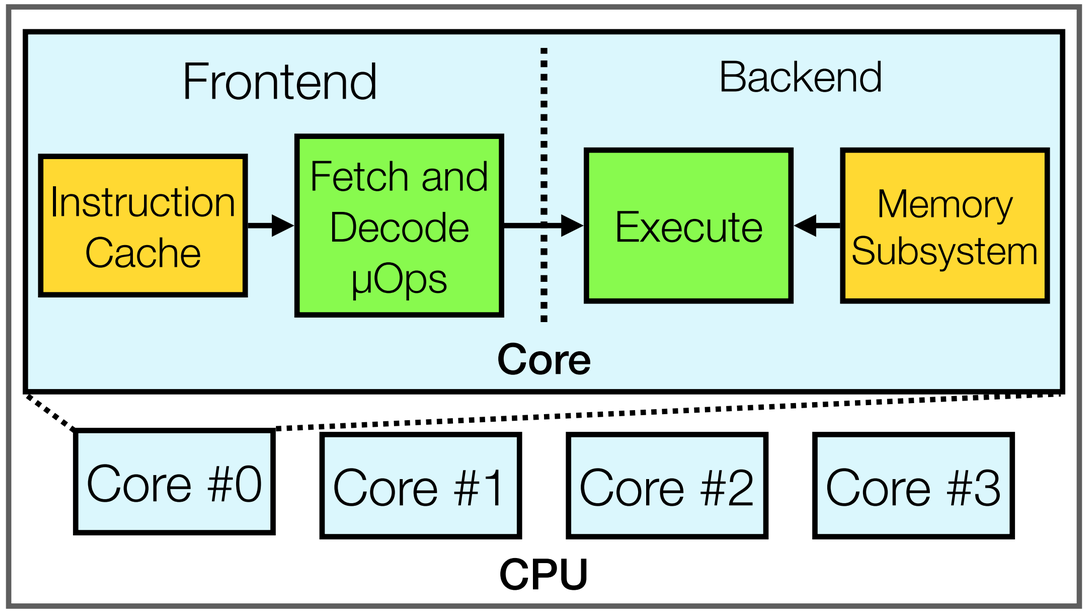
Micro-architecture Analysis



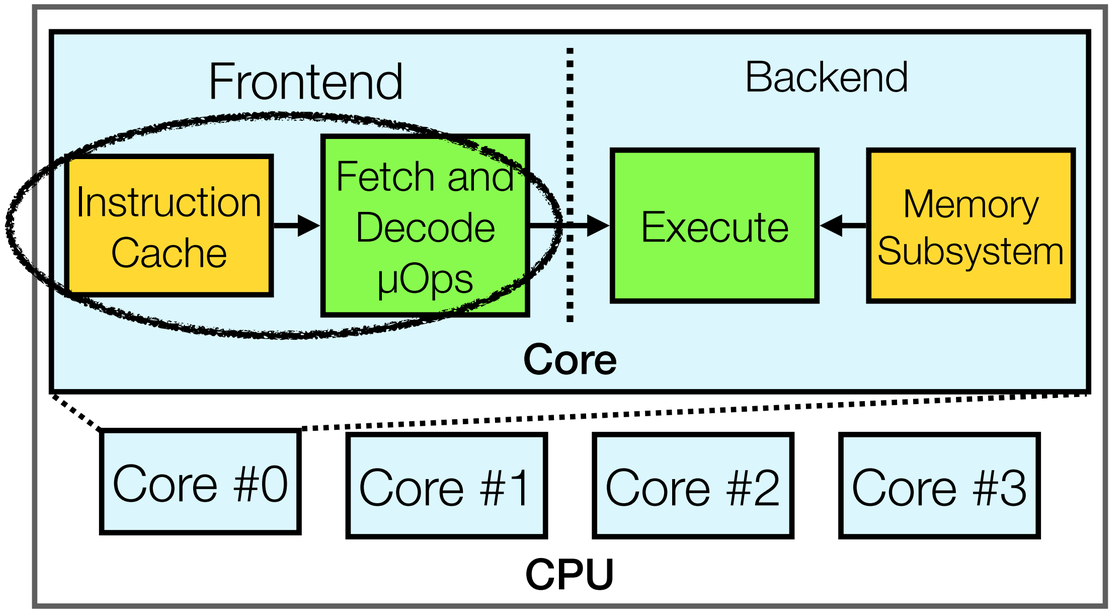
Micro-architecture Analysis



Micro-architecture Analysis

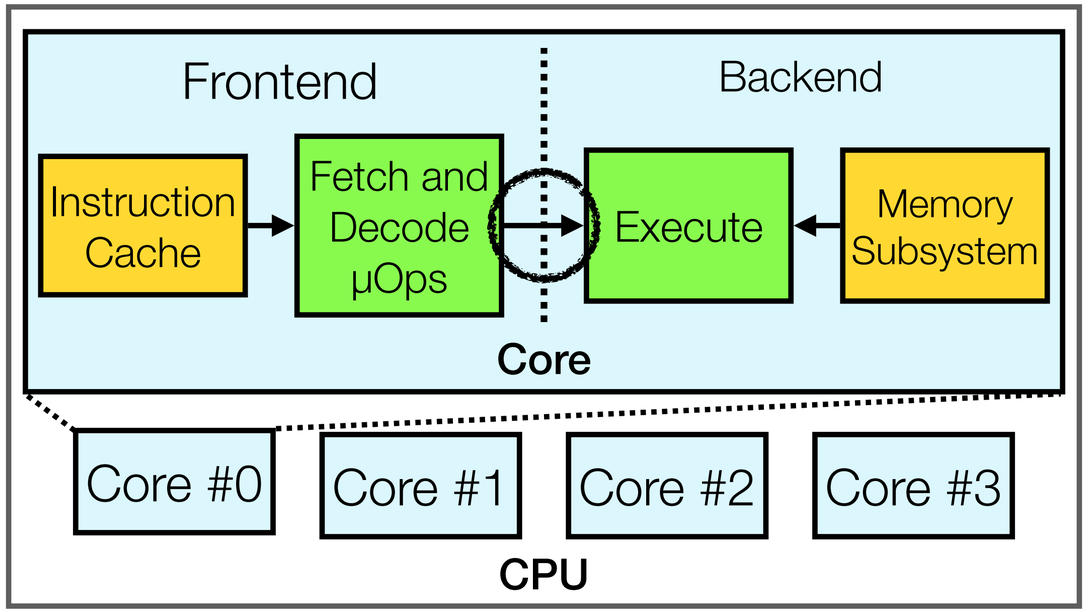


Micro-architecture Analysis



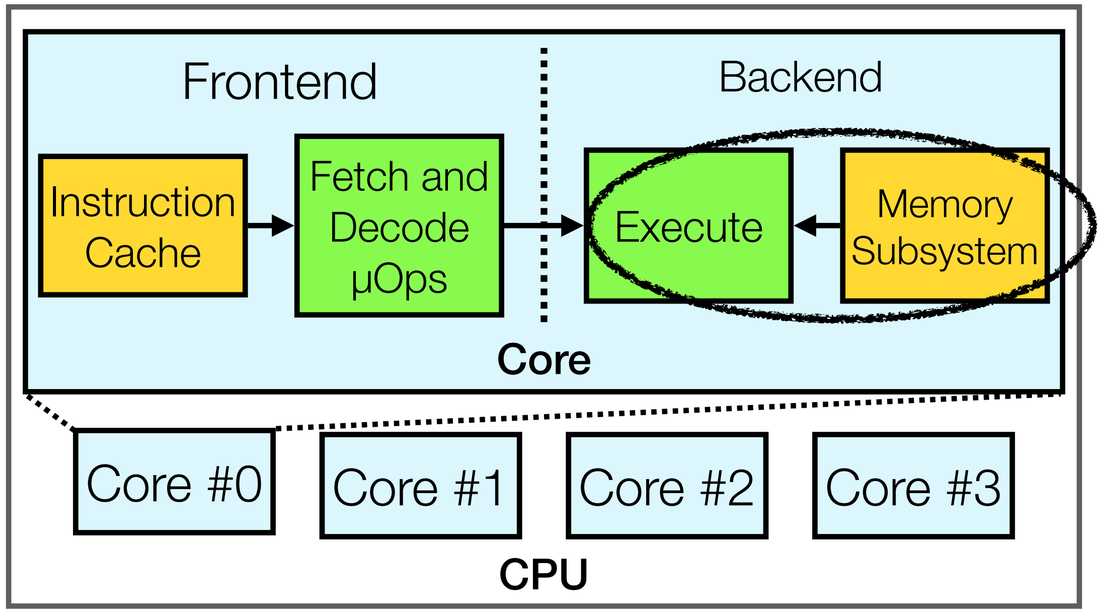
Complex instructions in L1i decoded in µOps

Micro-architecture Analysis



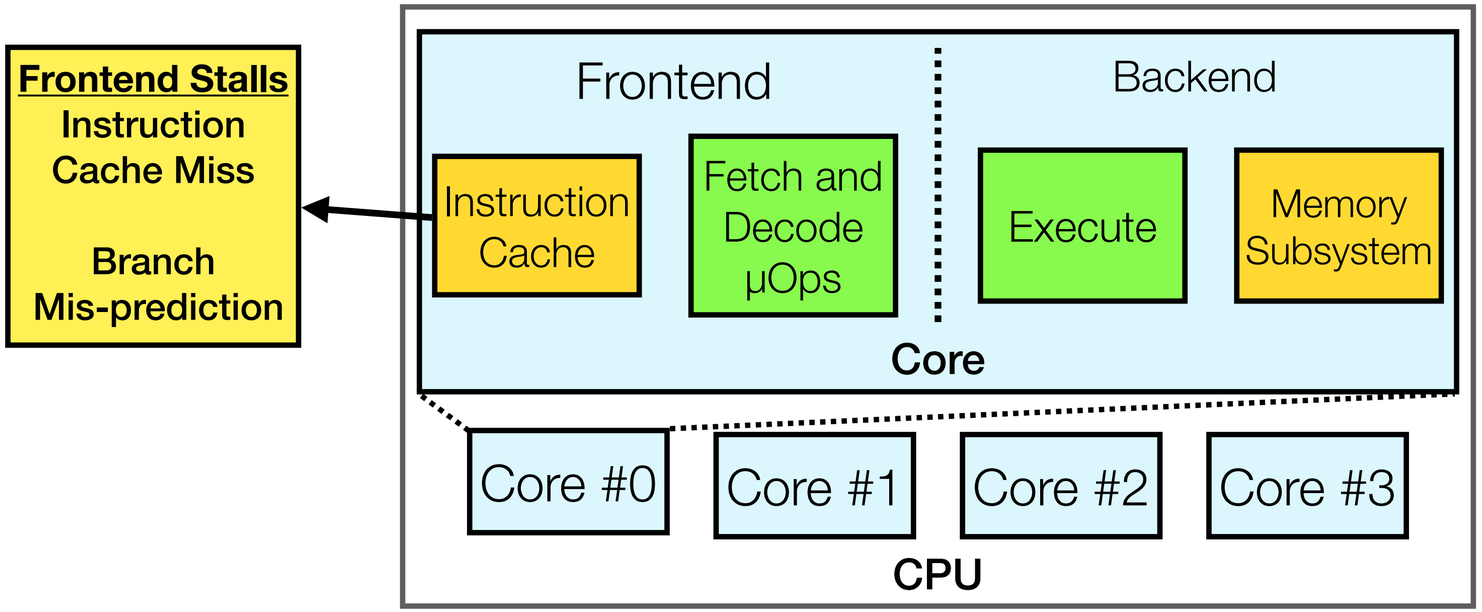
Frontend delivers up to 4 µOps per cycle to backend (Intel)

Micro-architecture Analysis

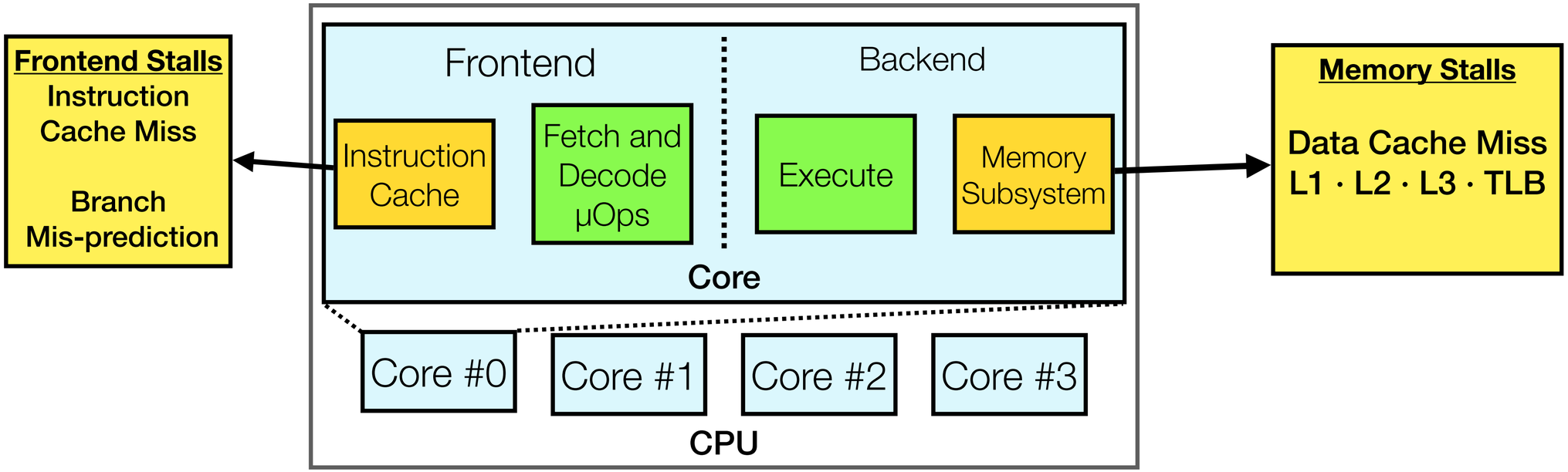


Provides data to registers from L1d, L2, LLC, and Main-Memory

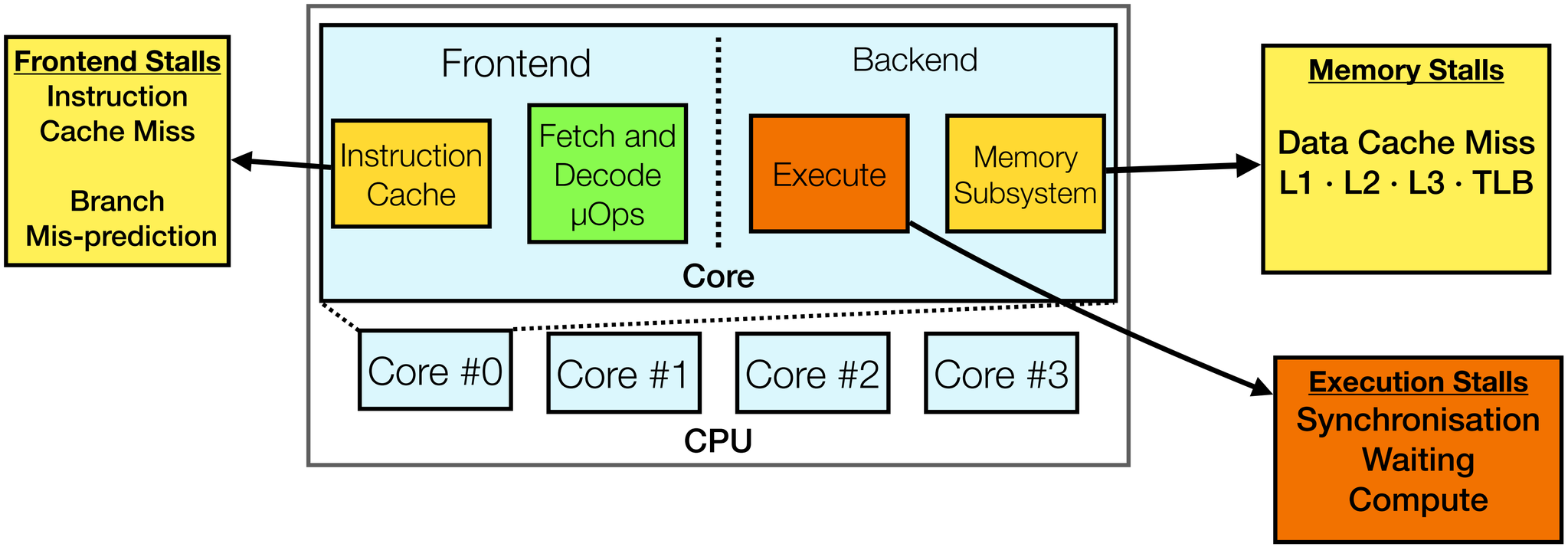
Micro-architecture Analysis



Micro-architecture Analysis

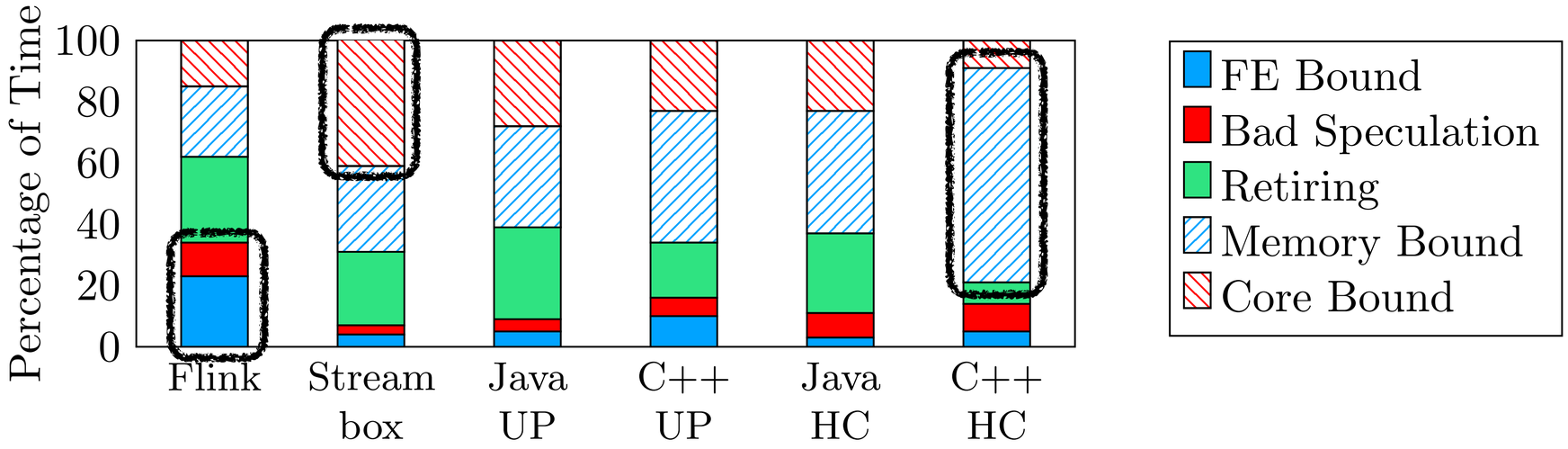


Micro-architecture Analysis



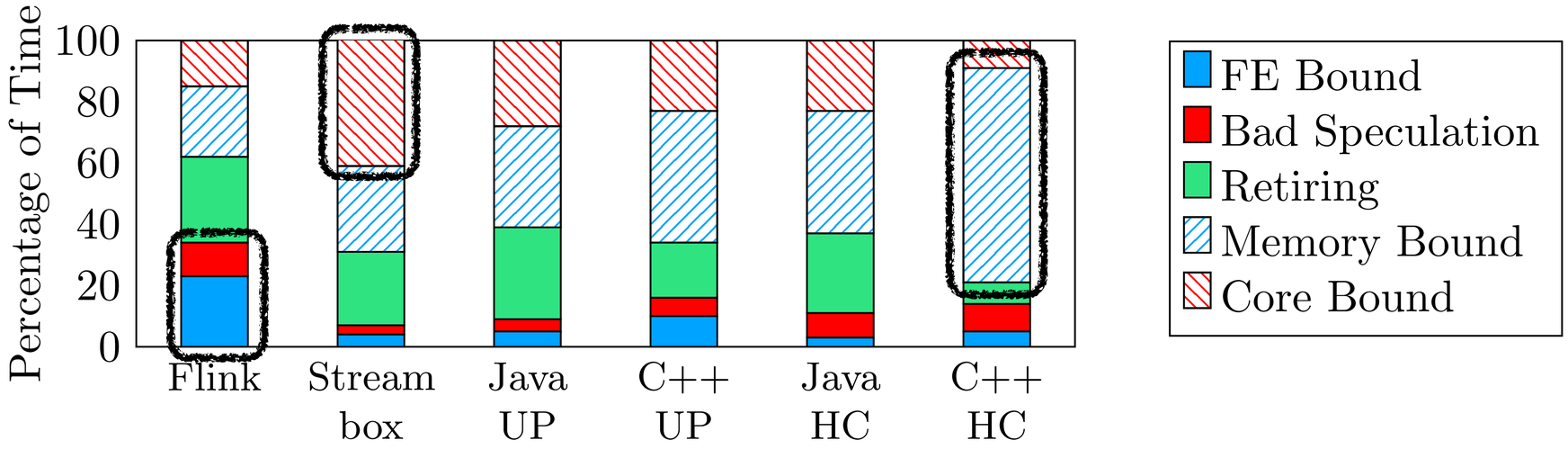
Hardware Performance Counters help us understand CPU performance

Inefficiency explained



Large instruction footprint, virtual functions, (de-)serialisation, and suboptimal data access pattern

Inefficiency explained



Large instruction footprint, virtual functions, (de-)serialisation, and suboptimal data access pattern

Poor data and code cache locality

When Query-Compilation makes sense

..over Interpretation-based vectorized query execution

- Always performance gain by removing virtual function calls, reducing code footprint, improves data locality (efficient memory access patterns)
- however, hard to maintain and debug and requires suitable frontend and IR
- UDFs are a problem
  - black-box: performance depends on UDF implementation
  - look inside the UDF to holistically optimise query: better but how?
  - UDFs with restricted semantics?

How to architect a streaming query compiler

- Do I need a query compiler?
- Define query language and semantics (embedded, dialect)
- Define IR and what to capture (transformation, side-effects, state)
- Latency of query compilation (full opt, JIT, copy-and-patch)
- Codegen to C++/Rust or LLVM IR or ..?
- Optimizing query compiler? Use live-statistics and keep optimising

When LM/GM make sense

- Cost(Partitioning) > Cost(LM or GM)
- LM outperforms GM when partitioning keys follow a skewed distribution
  - no conflicts but LM requires multiple merging steps: Cost(Merging)<Cost(Conflicts)
- GM is suitable with uniform distribution (see Grizzly)

Spark DStream Tuning

- reduceByKeyAndWindow and CustomReceiver
- Followed best practices available in 2018
- Had to figure out spark.streaming.receiver.maxRate
- No disk storage or compression
- G1GC

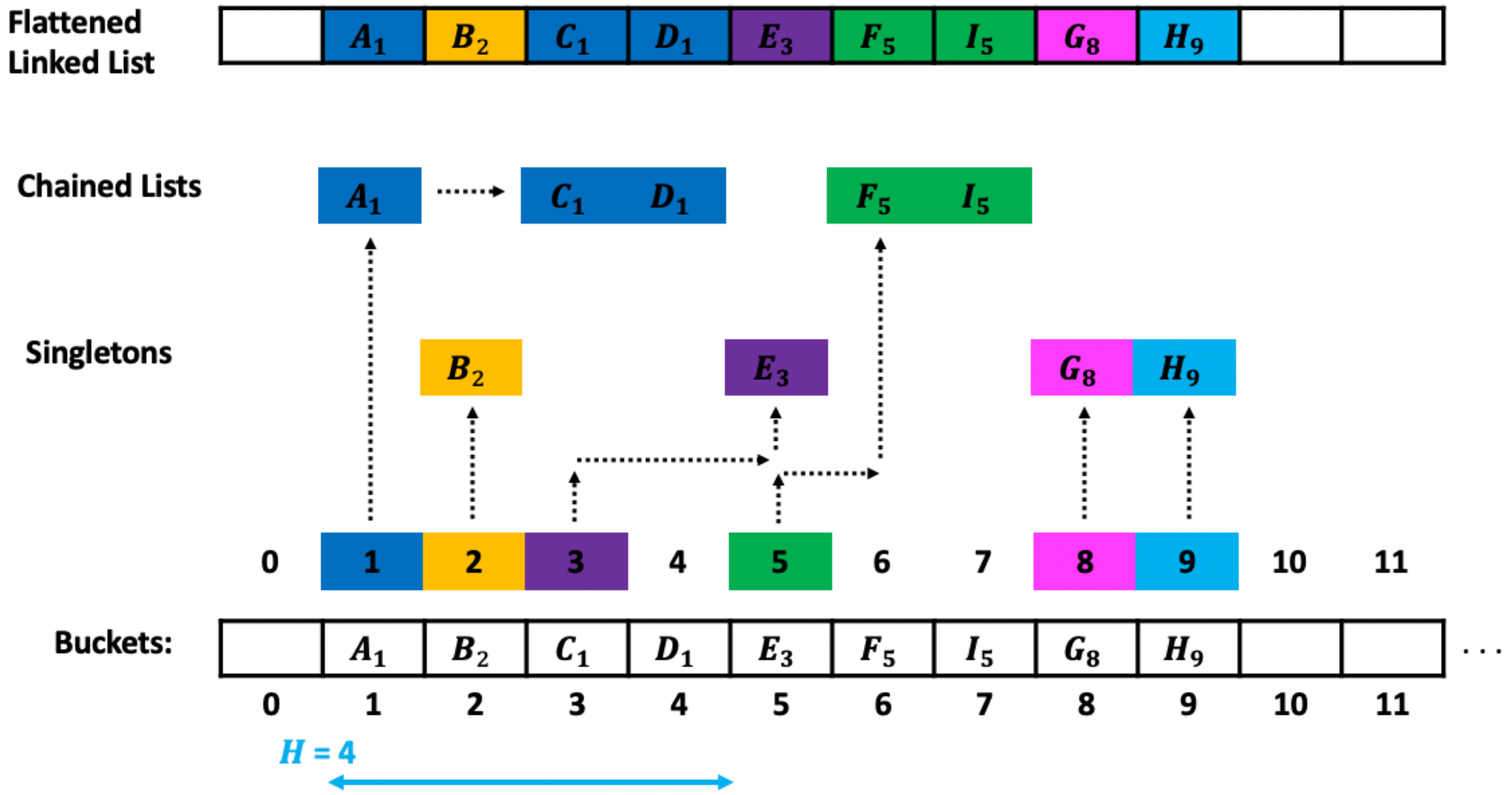
Flink Tuning

- Followed best practices available in 2018
- Custom (de-)serializers
- Disable checkpointing
- G1GC

Outlook: improve state management

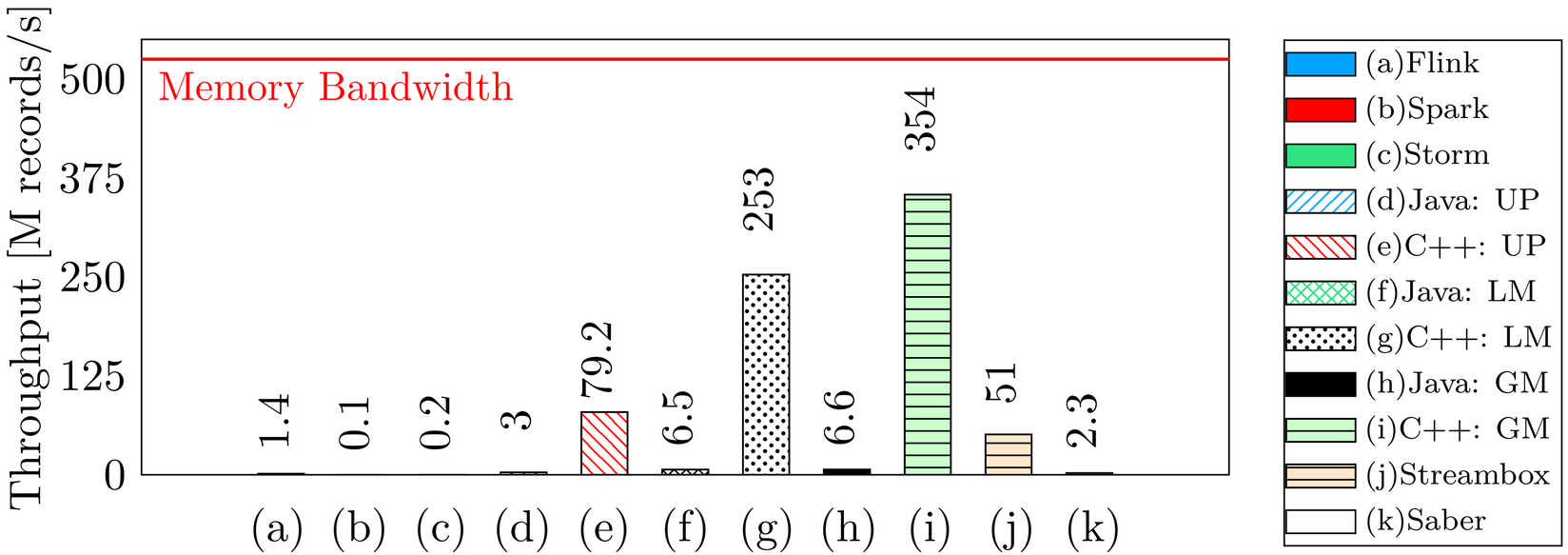
- In-memory hash-tables or LSM-Trees that neglect streaming semantics
- Not even a problem when in JVM due to impedance mismatch with C++ impl.
- Research outlook: consider streaming-aware storage
  - Temporal and spatial locality of state access
  - Design for modern-hardware: cache-friendly, local storage, remote storage
  - Perform GC at window boundaries
  - Make fault-tolerant (e.g., Scabbard)

Hopscotch Hashing

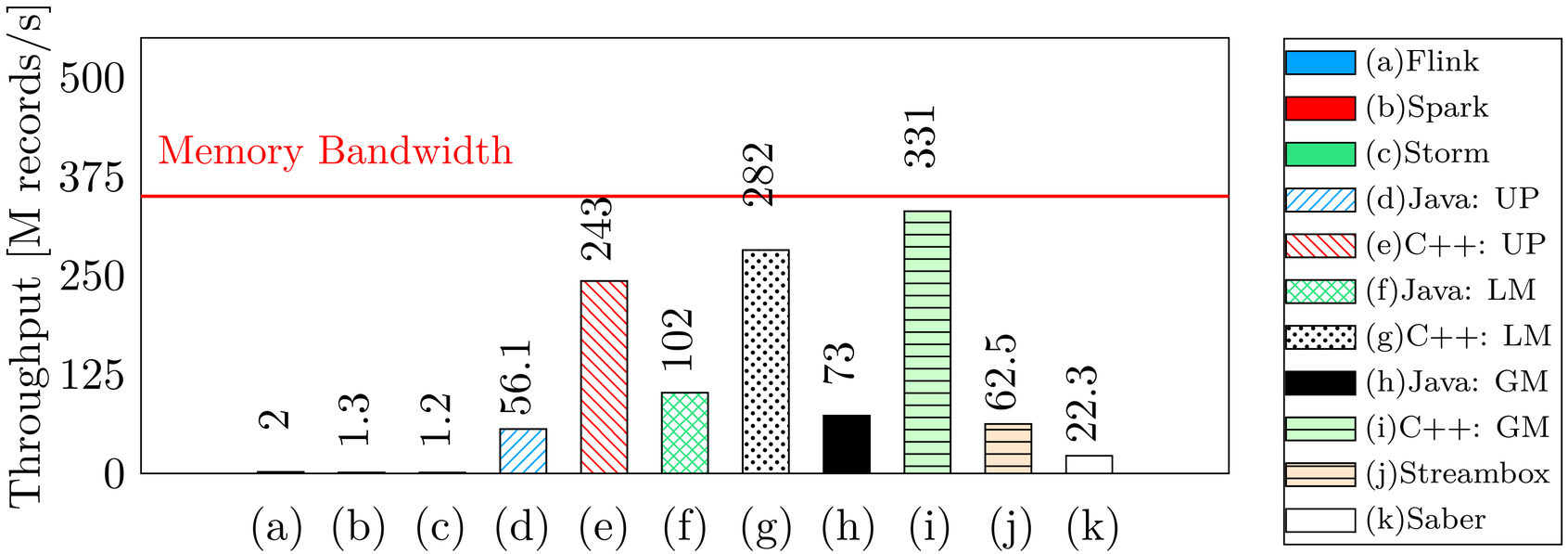


Open Addressing: it uses H neighbouring (consecutive) buckets for each bucket
Invariant: cost of finding item in neighbourhood = cost of finding item in the exact bucket

LRB - Toll and accidents

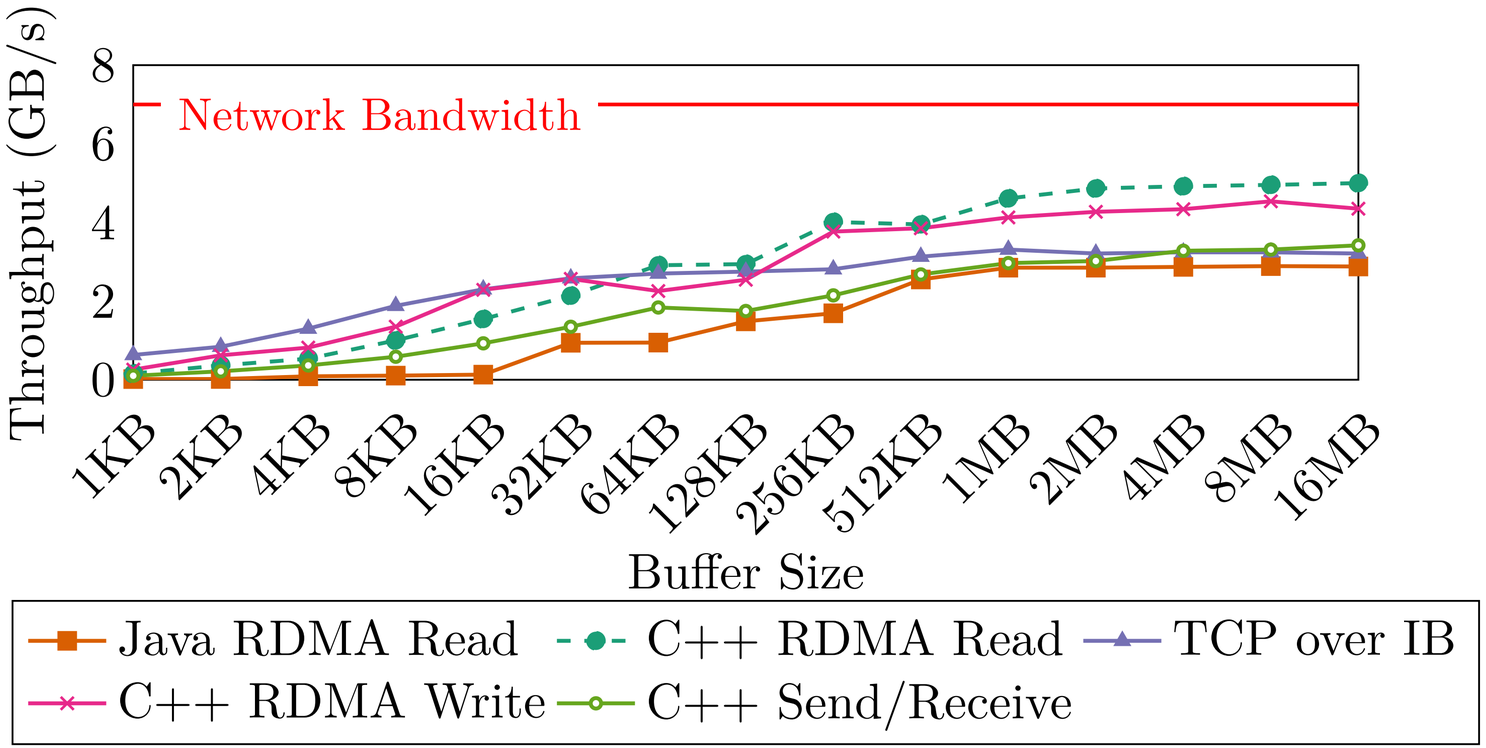


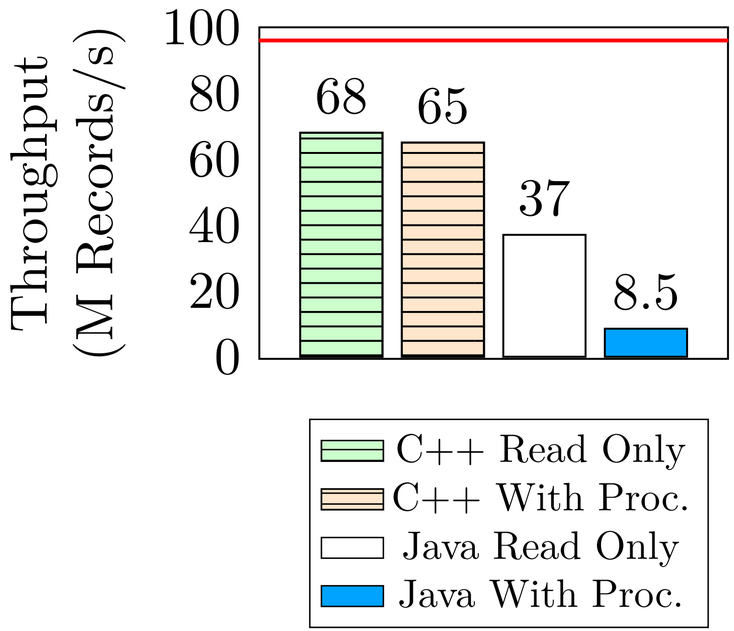
NYT



What are the number of trips and their average distance for the VTS vendor per region for rides more than 5 miles over the last two seconds?

Early RDMA Benchmarks



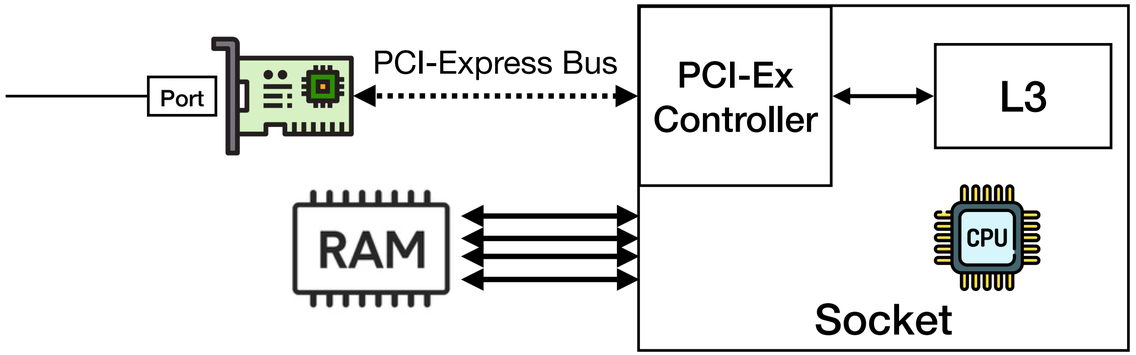


Backup

Slash

Remote Direct Memory Access

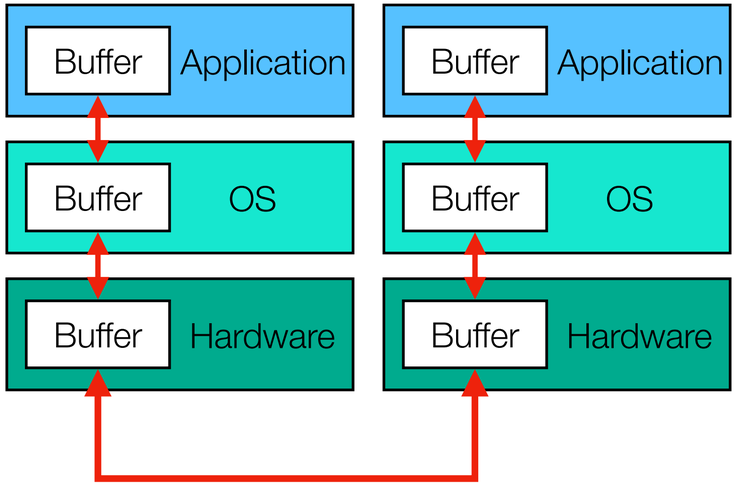
Infiniband EDR 100Gbps (12.5 GB/s)
Infiniband HDR 200 Gbps (25 GB/s)
Infiniband NDR 400 Gbps (50 GB/s)



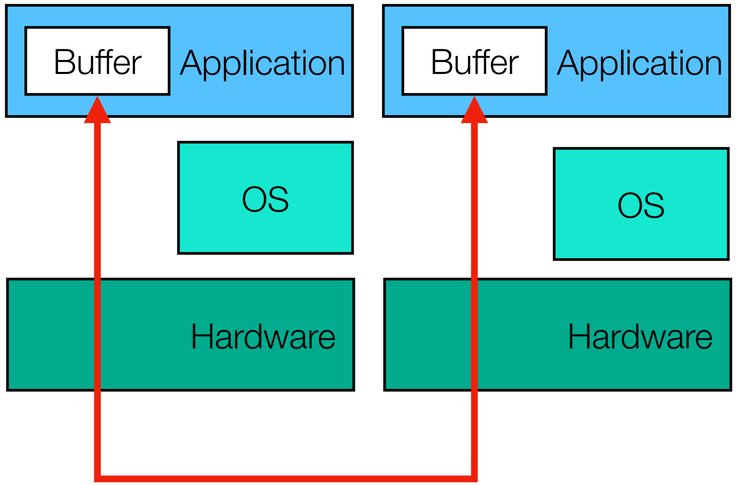
PCI-Express 3.0 Bandwidth: 984.6 MB/s per lane (16x: 15.74 GB/s)

PCI-Express 5.0 Bandwidth: 3.93 GB/s per lane in each direction (16x: 63 GB/s)

Socked-based vs. RDMA



Kernel-space Networking
Multiple Data Copies



User-space Networking (low latency)

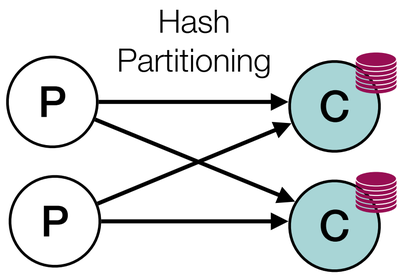
Zero-copy (low latency)

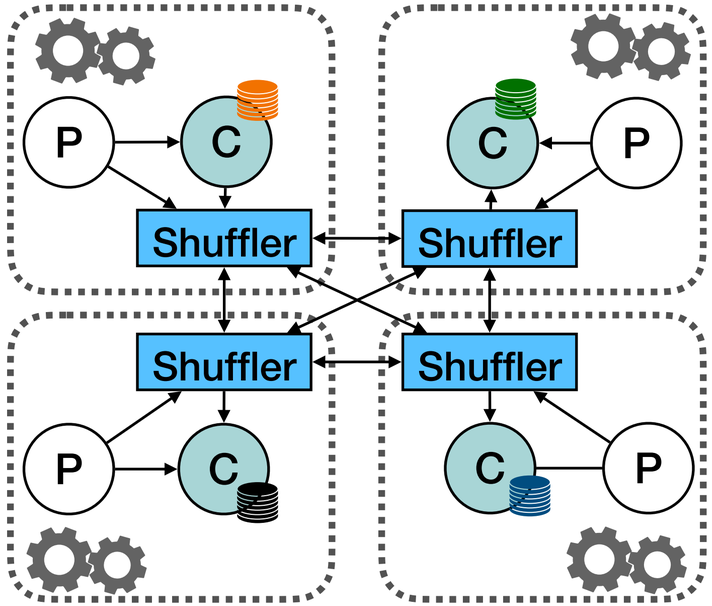
Enables distributed memory (non coherent)

Two-sided verbs: Send/Recv
One-sided verbs: Read/Write/Atomic

Distributed Streaming Query Execution

Partitioning-based Execution

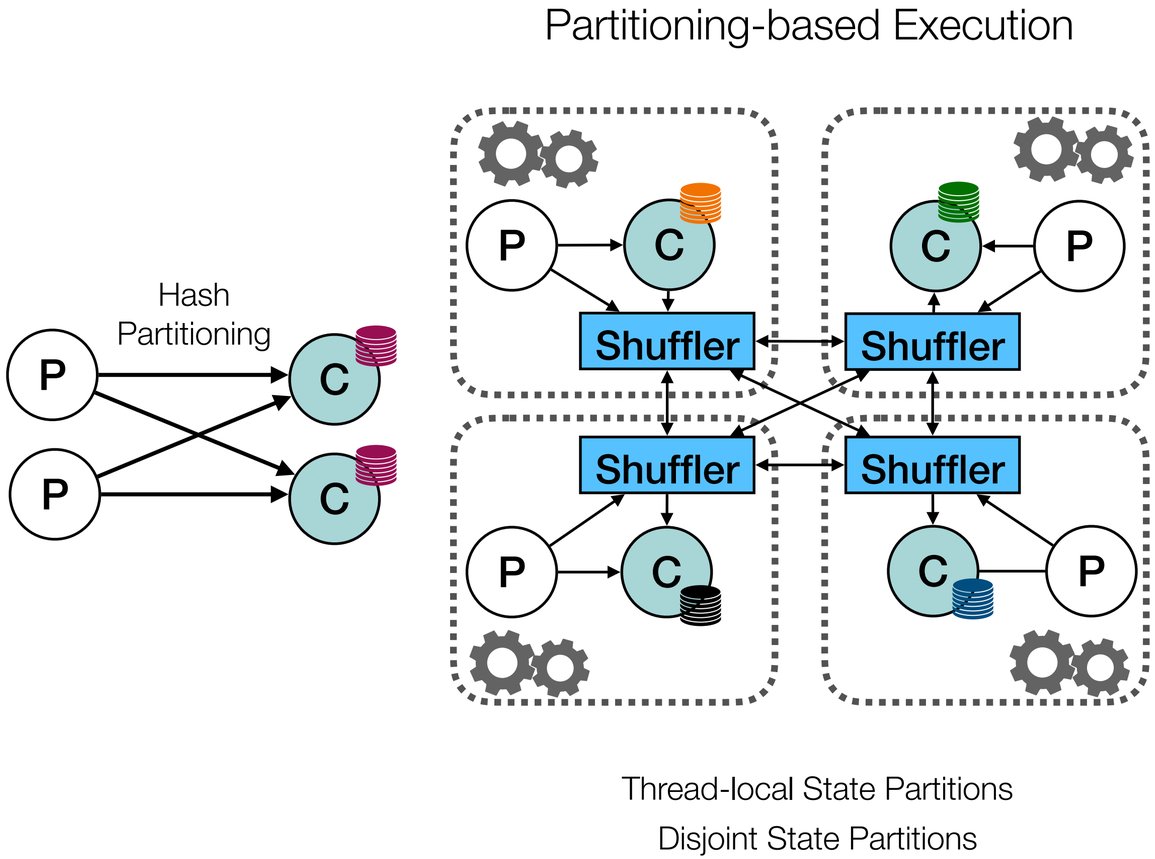


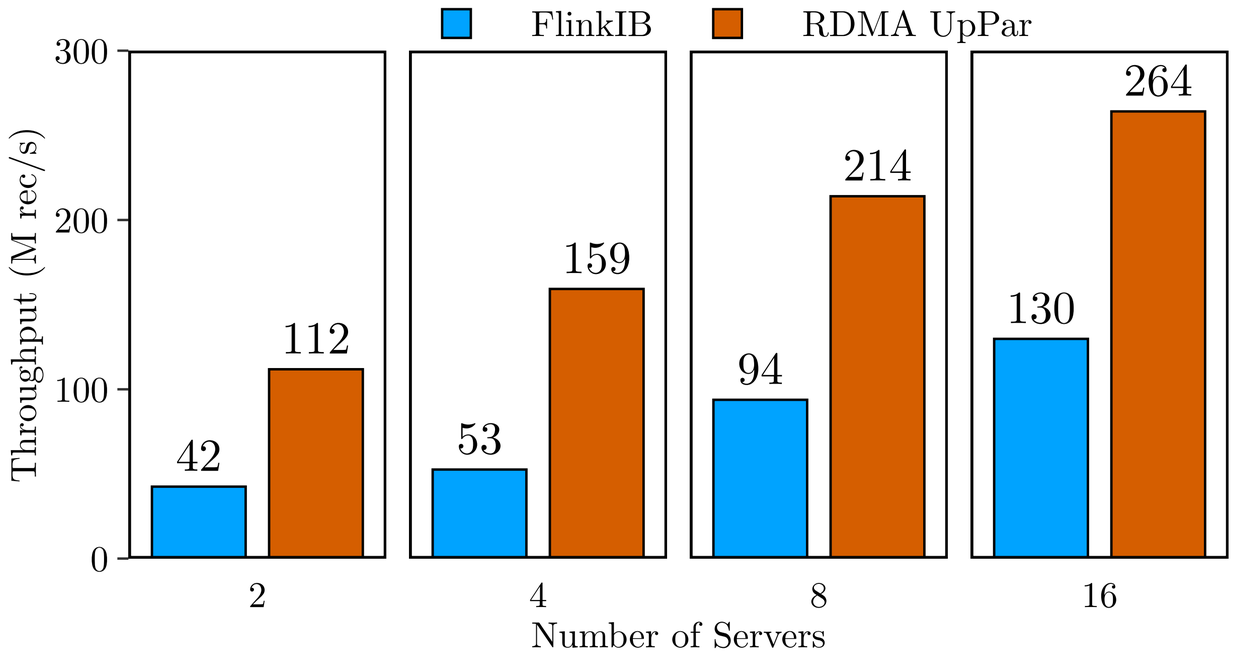


Thread-local State Partitions

Disjoint State Partitions

Distributed Streaming Query Execution



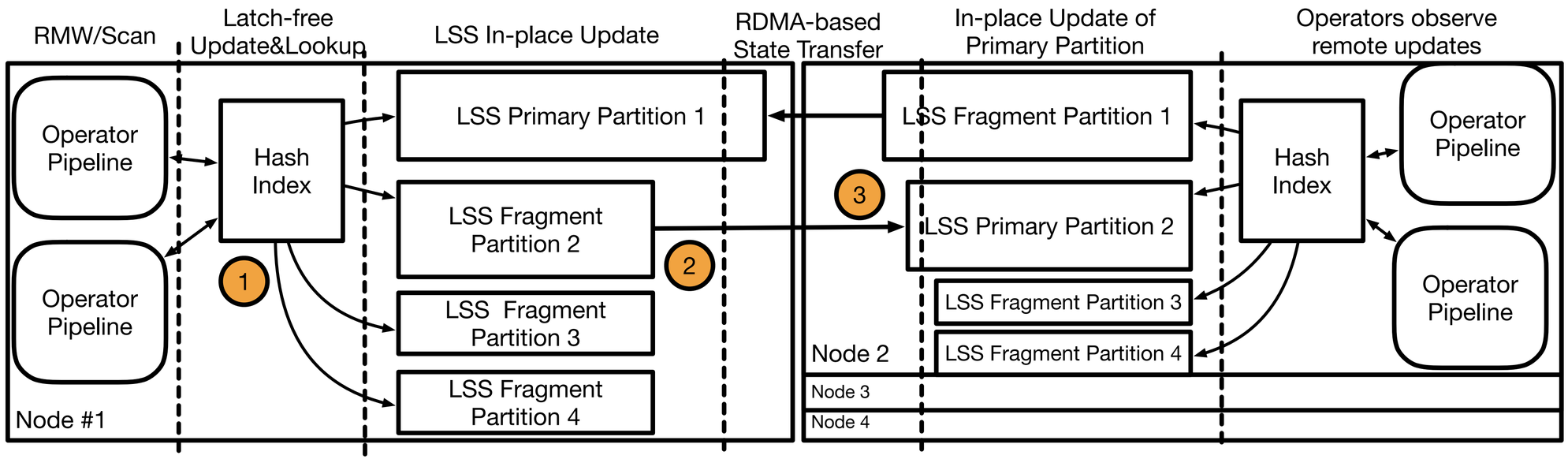


Intel Xeon Gold 5115 @ 2.4 Ghz 10-cores
L1: 32KB L2: 10MB
L3: 13.75MB RAM: 96GB
NIC: Mellanox Connect-X4 EDR 100Gbps

When Slash make sense

- Cost(Partitioning) + Cost(Local Computation) > Cost(Partial Computation) + Cost(Lazy Merge)
- Keyed Aggregation or Joins (Streaming ETL)
  - Define State as a CRDT
- New operators need to use our distributed state abstraction
  - Network-hungry such as Cross-Product
  - ML Operators

Where RDMA comes into play



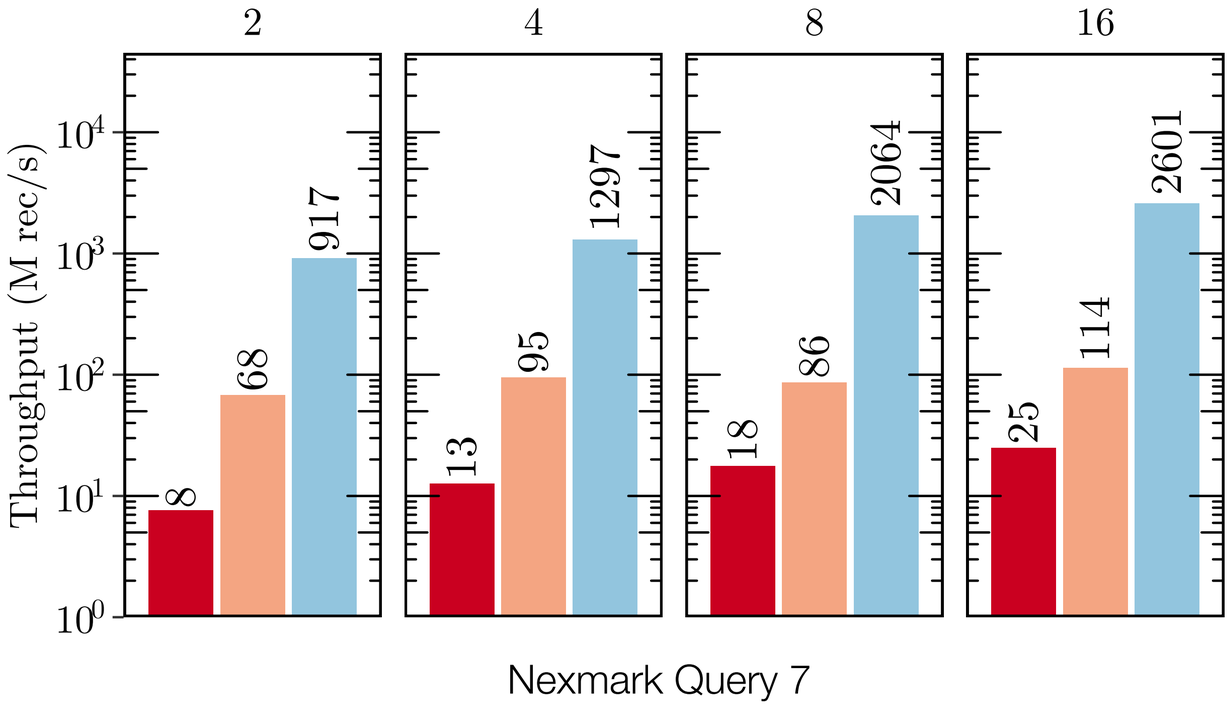
Cost of RDMA

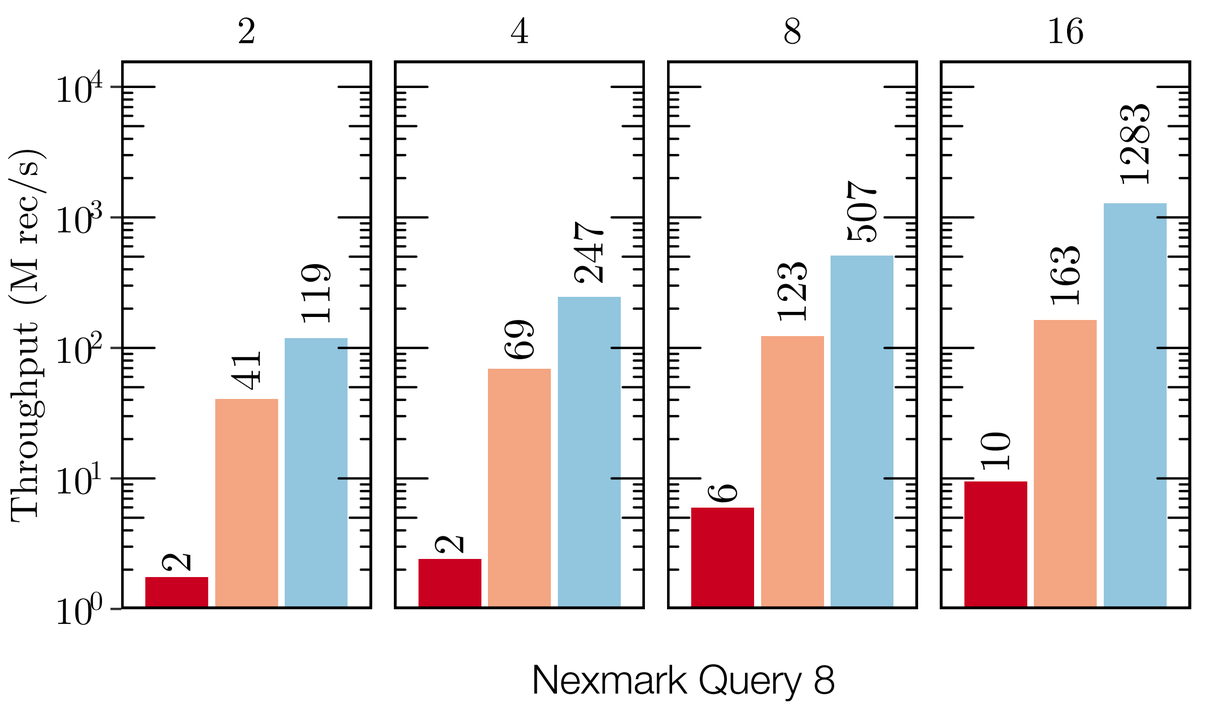
- Mellanox (now Nvidia) Connect X-6 200Gpbs sold at about 1200$
- Azure RDMA-capable H/HB instances: 800/1600$/mo
- AWS has Elastic Fabric Adapter (Send/Recv): 2180$/mo (m6in.32xlarge)

Large SPE deployments

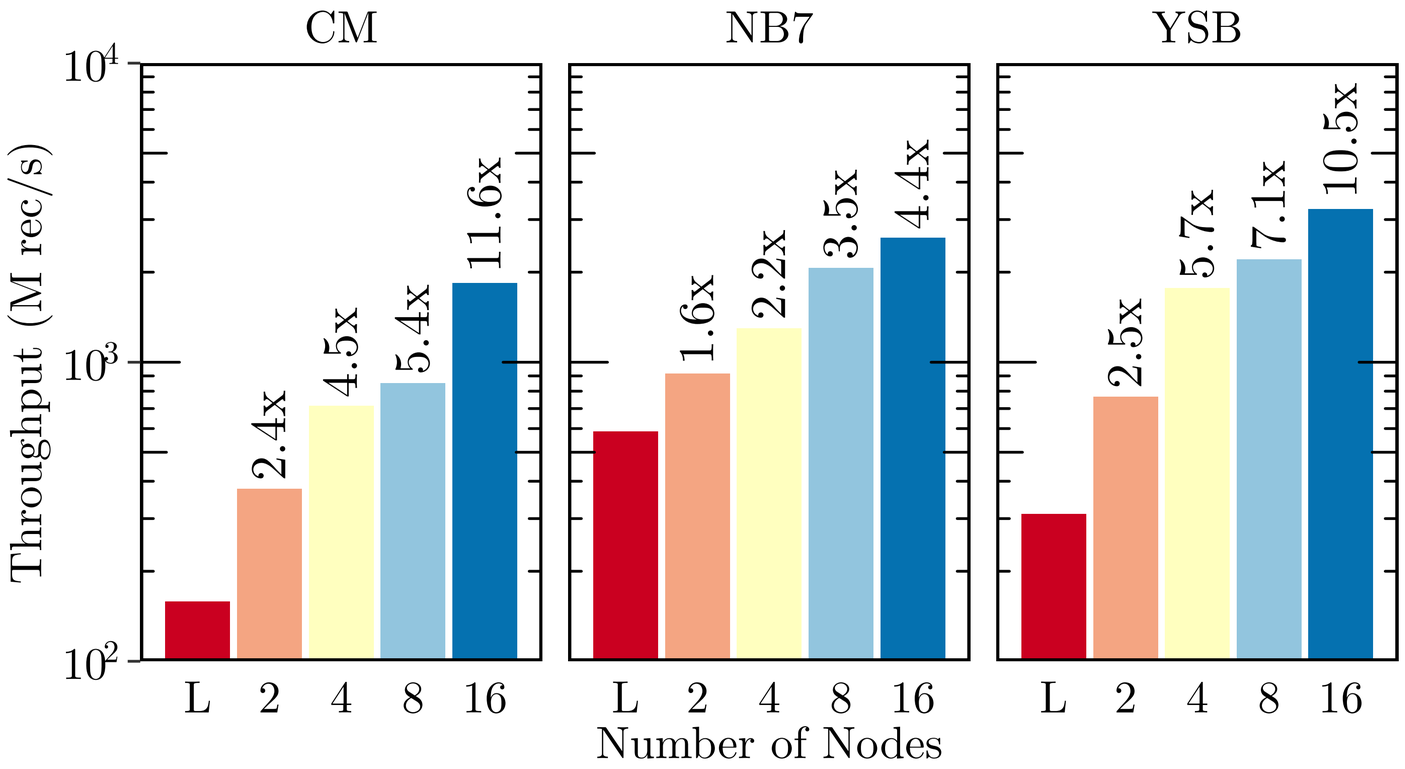
- Alibaba: 1.5M CPU for Flink (35000 jobs)
- Netflix: 14k nodes with 22k CPU (100s jobs)

Slash Performance

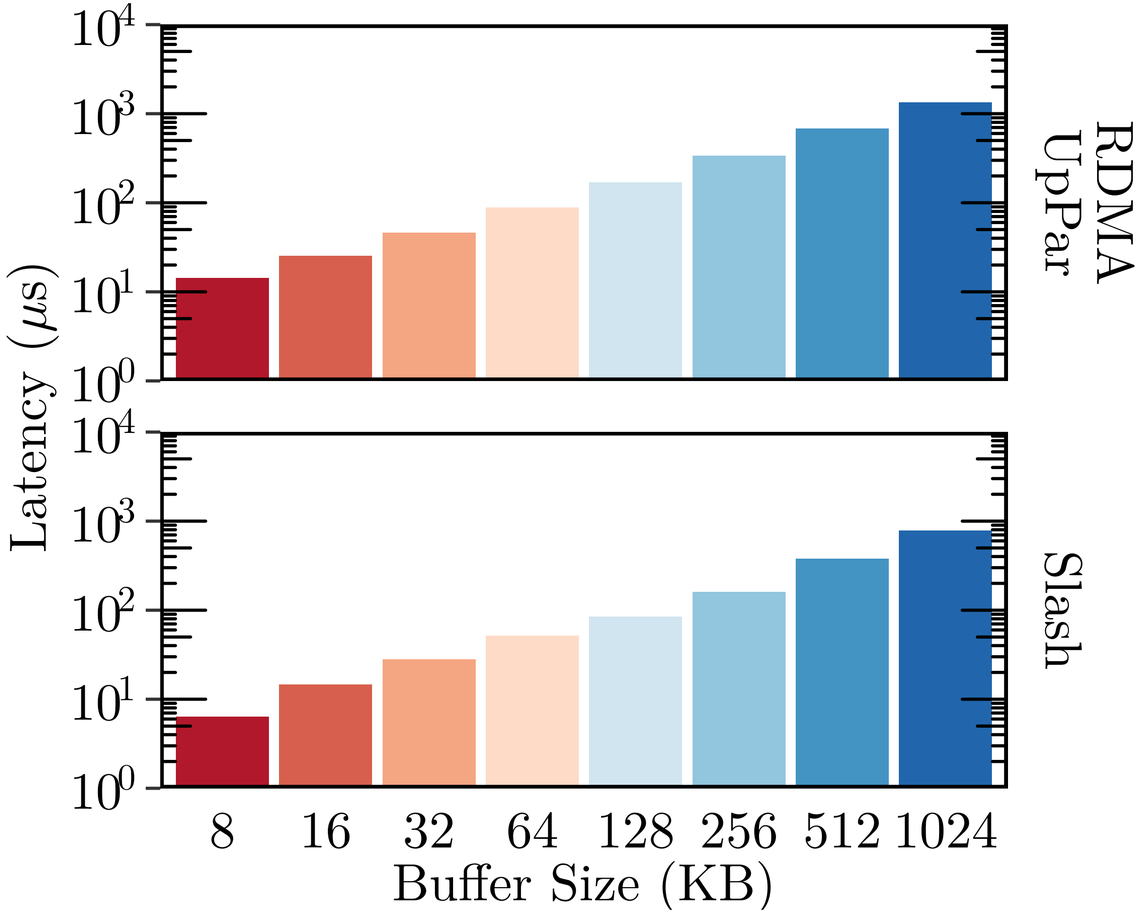




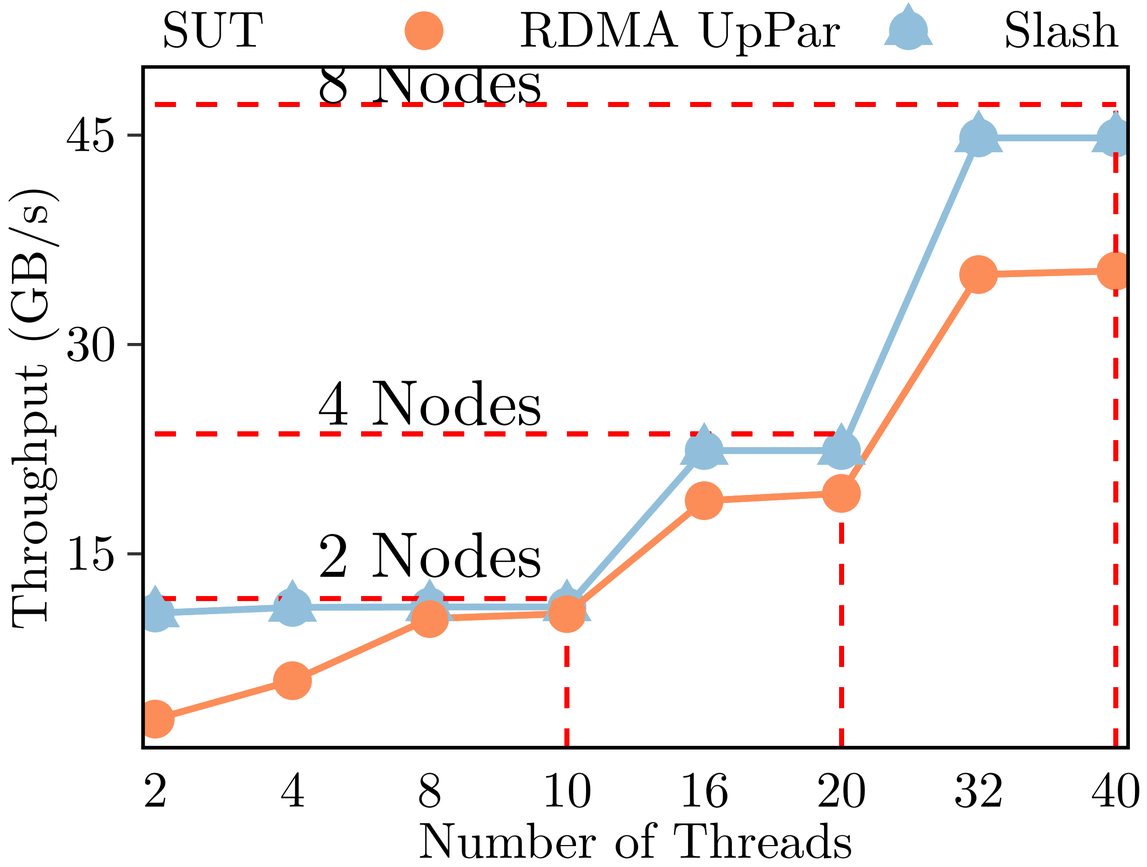
Slash Microbenchmarks: COST



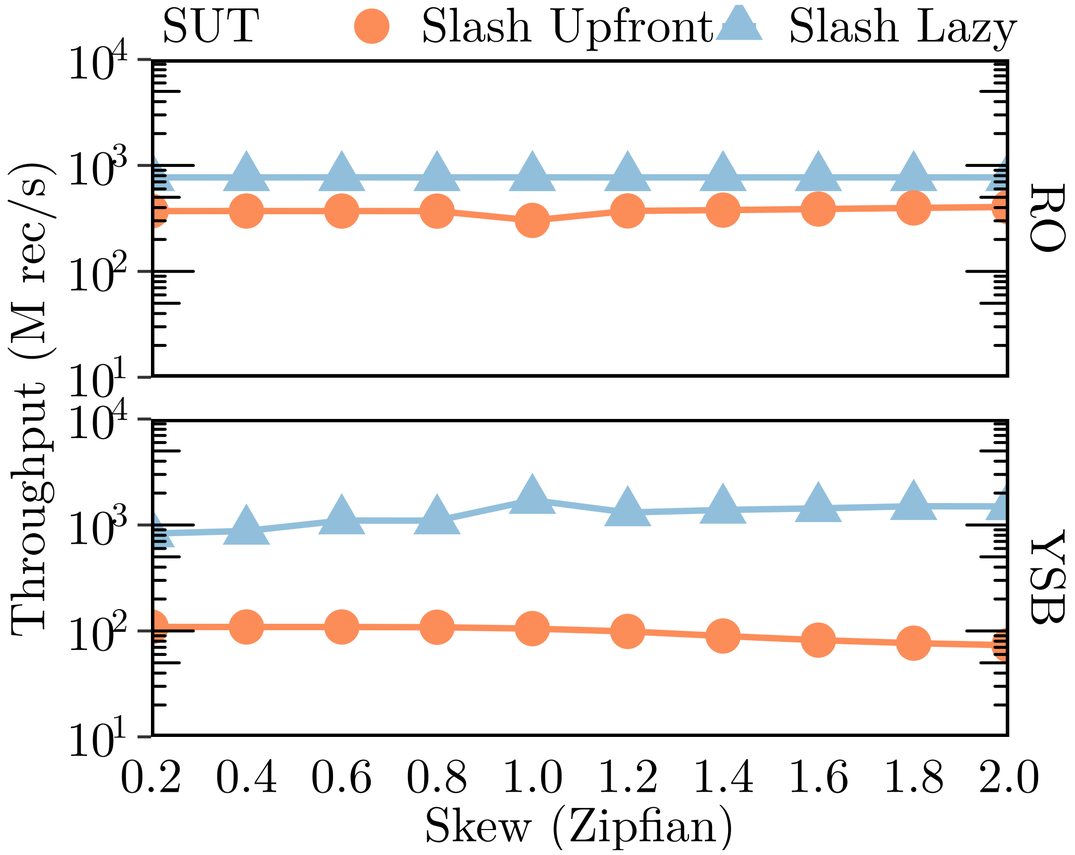
Slash Microbenchmarks: Latency



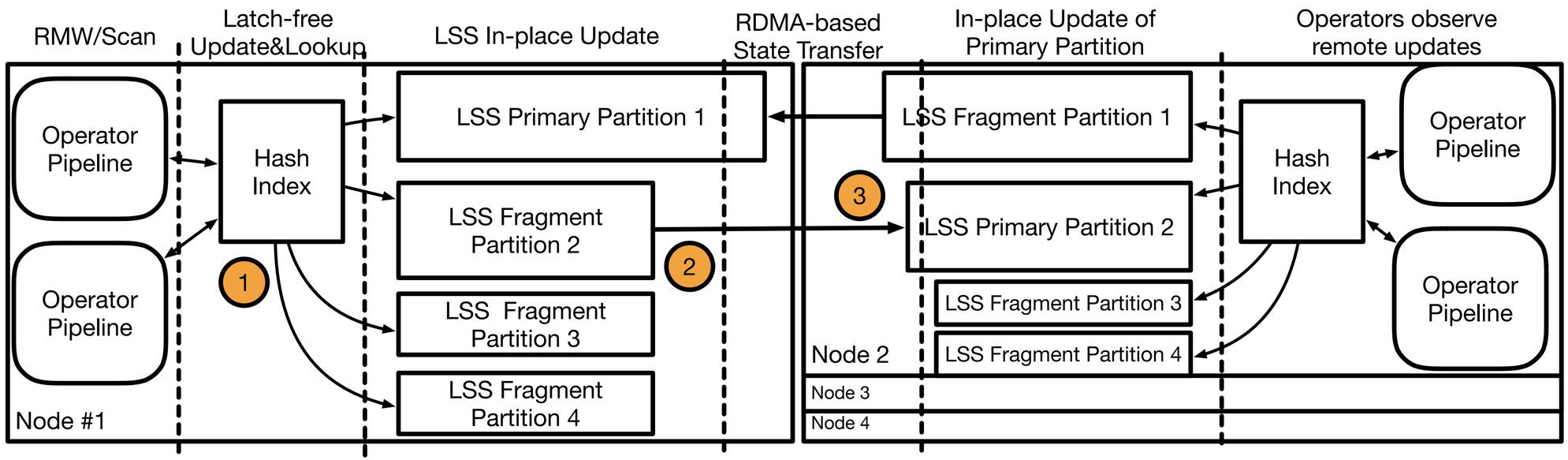
Slash Microbenchmarks: Node Parallelism



Slash Microbenchmarks: Skew

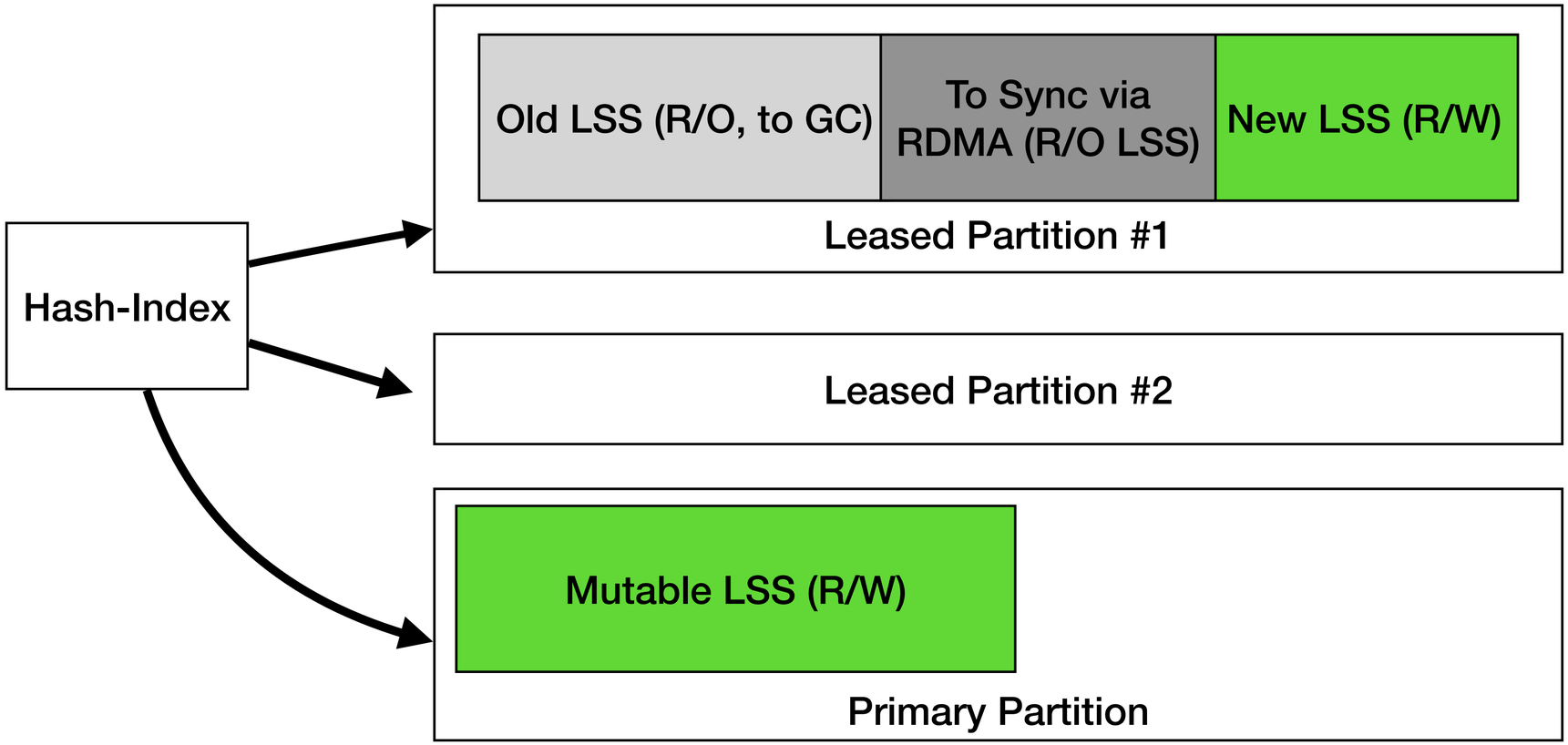


Slash State Backend Internals



Fragment Partition (thesis)
is the Leased Partition (talk)

Anatomy of Slash Partitions



Conflict Free Replicated Data Types

- Inspired by AnnaKVS and FASTER design
- Define a “merge function” *f(k, v1, v2)* to merge v1 and v2 within the same window
- Windowed aggregation:
  - Average, Sum, Count
- Windowed Join:
  - List of segments

RDMA Data Channel details

- Pipelined RDMA Writes of data chunks arranged in a circular queue
  - Keep the RNIC well-fed with data
  - Async: too little -> low bandwidth; too much -> RNIC cache trashing
- Polling on footer
- Zero-copy
- Credit-based flow control to avoid producer overwhelm consumer

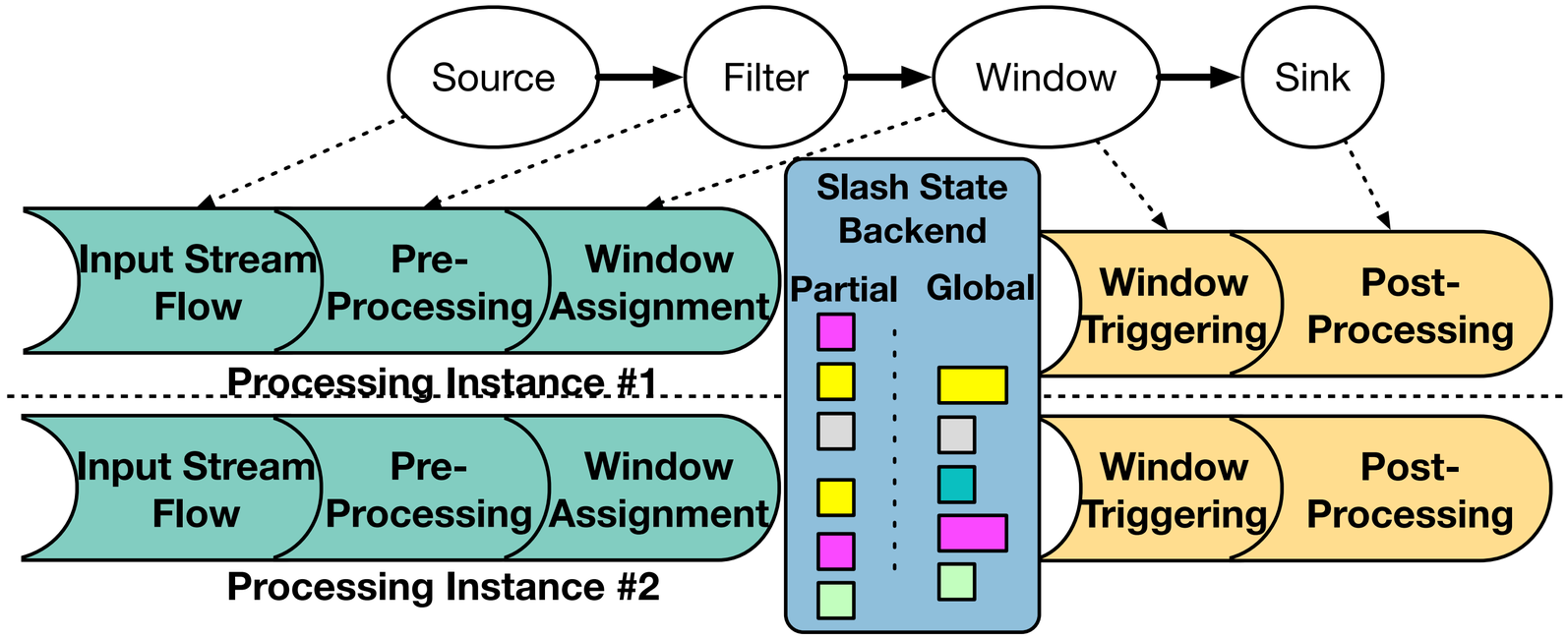
Going beyond rack-scale

- Slash requires a number of RDMA connections quadratic in the num of nodes
- Use Two-sided (Send/Recv) instead of RDMA Write/Read
  - Kalia et al.: RDMA requires NIC-managed connection state (a Connect-X5 RNIC drop 50% throughput with 5000 connections = 70 Slash instances)
  - RNIC SRAM: ~2 MB for connection and data structures, connection state ~375 bytes
  - Switch to application-managed connection state (datagram)
  - Requires software Congestion Control (e.g., rate-based) and achieves 70-92% of network throughput

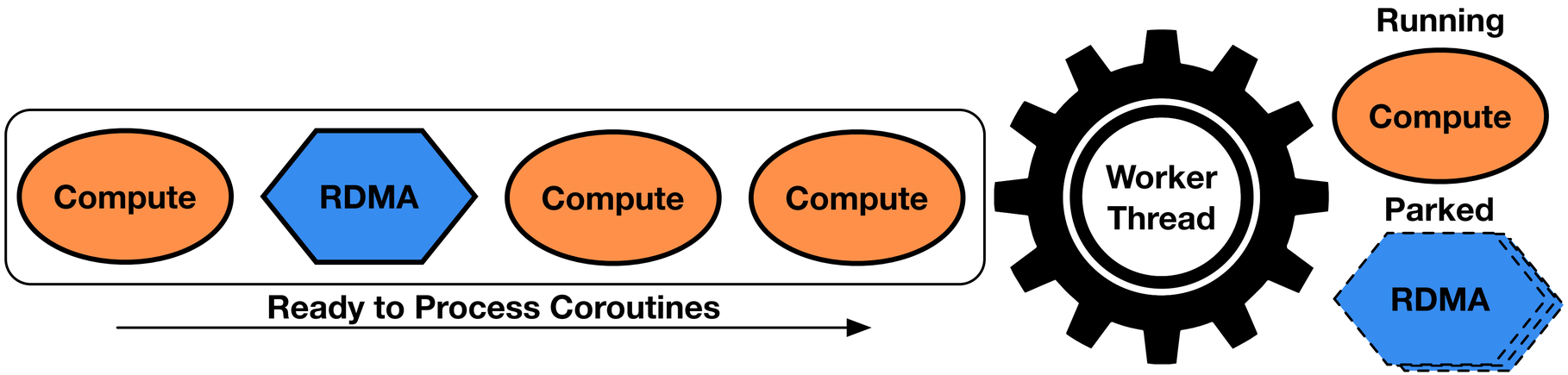
RDMA Atomics

- Bound by PCI-Ex RTT as a lock in the RNIC is held until the op is completed
  - 100s of ns with PCI-Express 3.0
  - Should evaluate with PCI-Express 5.0 and newer models?
  - Atomic semantics are atomic only among RNICs not CPU
  - Consensus in the Network community on avoiding them

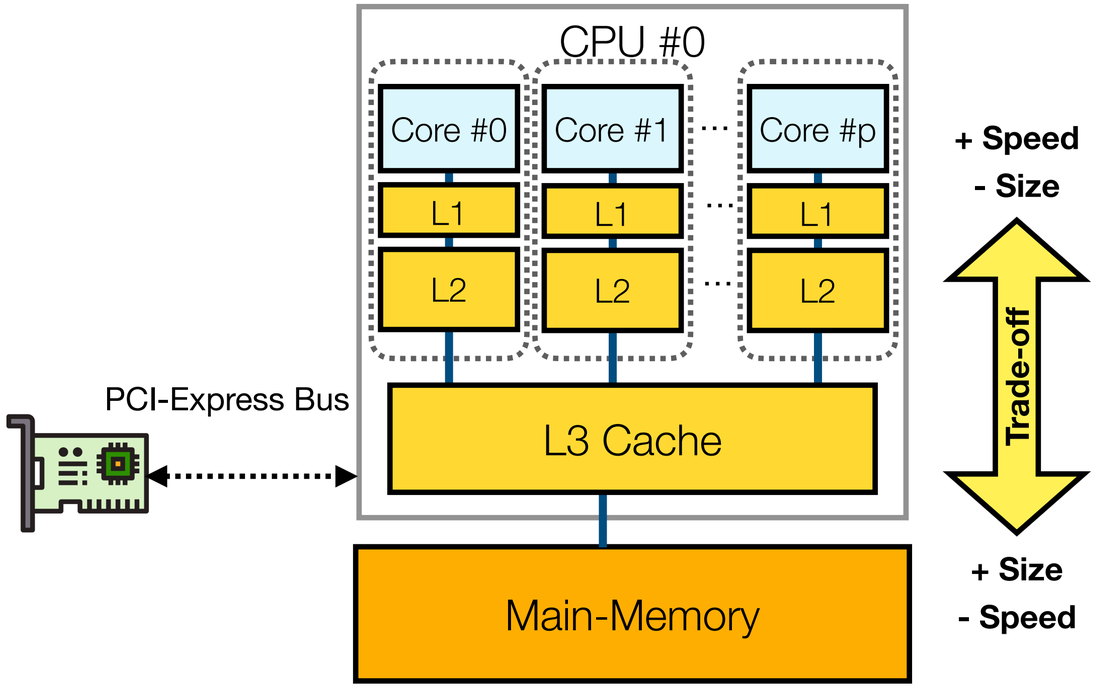
Slash internal processing



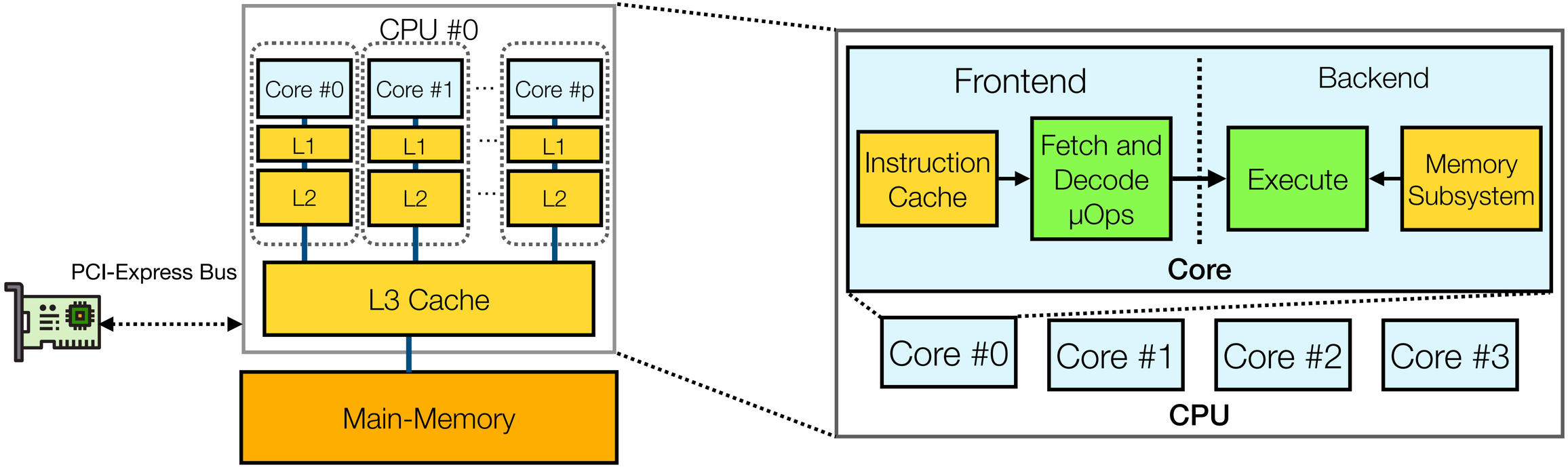
Slash internal processing



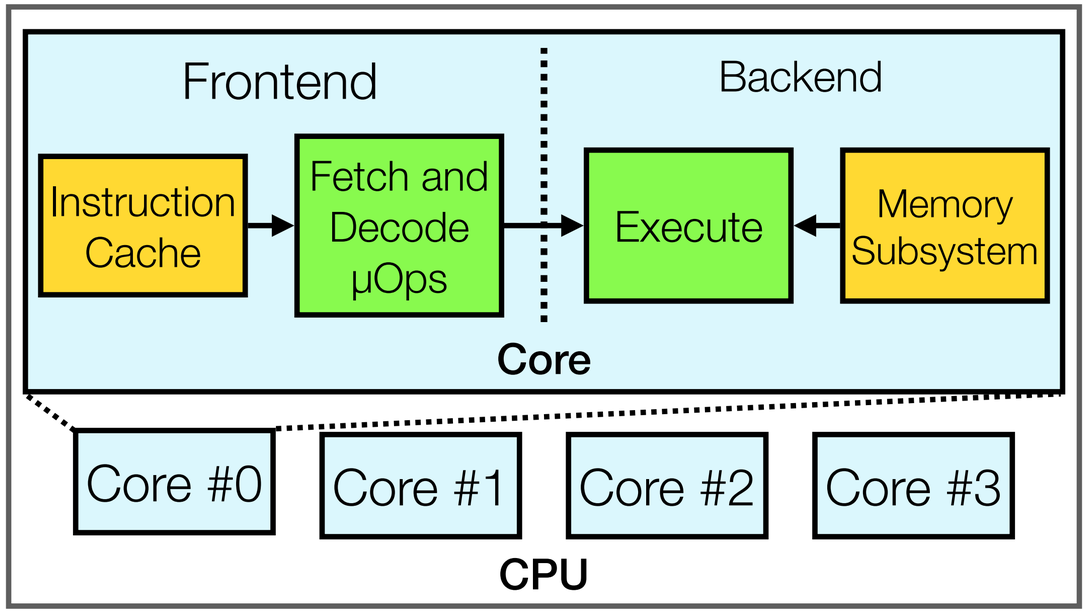
Micro-architecture Analysis



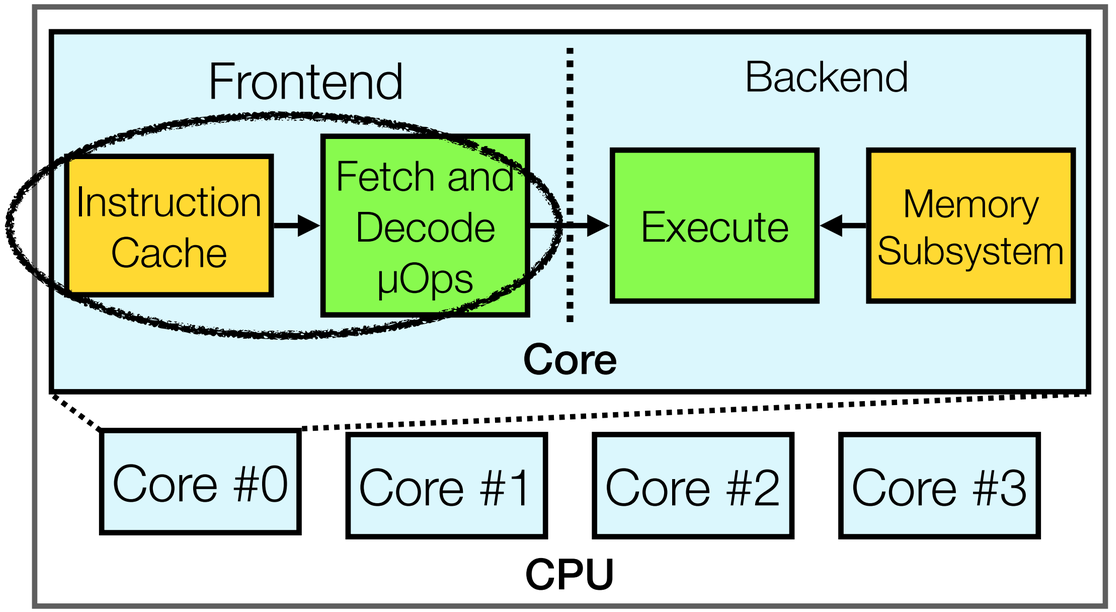
Micro-architecture Analysis



Micro-architecture Analysis

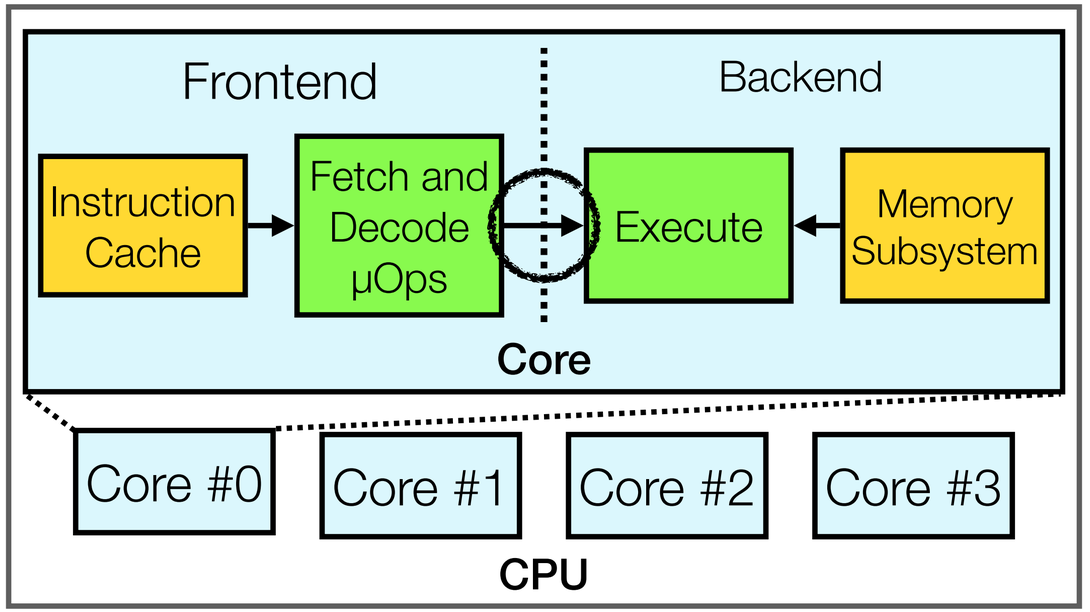


Micro-architecture Analysis



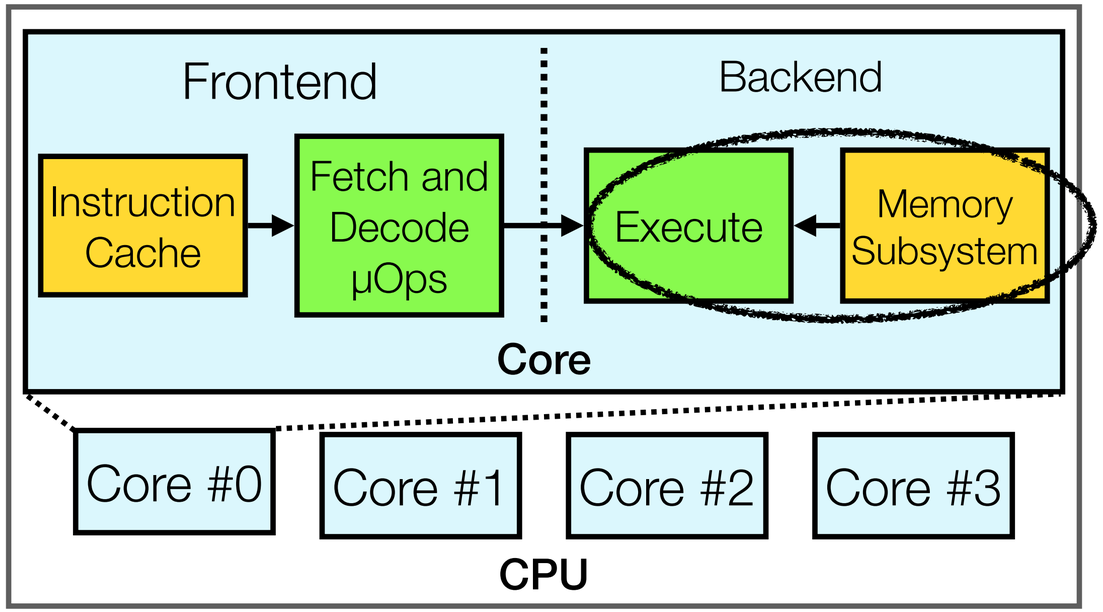
Complex instructions in L1i decoded in µOps

Micro-architecture Analysis



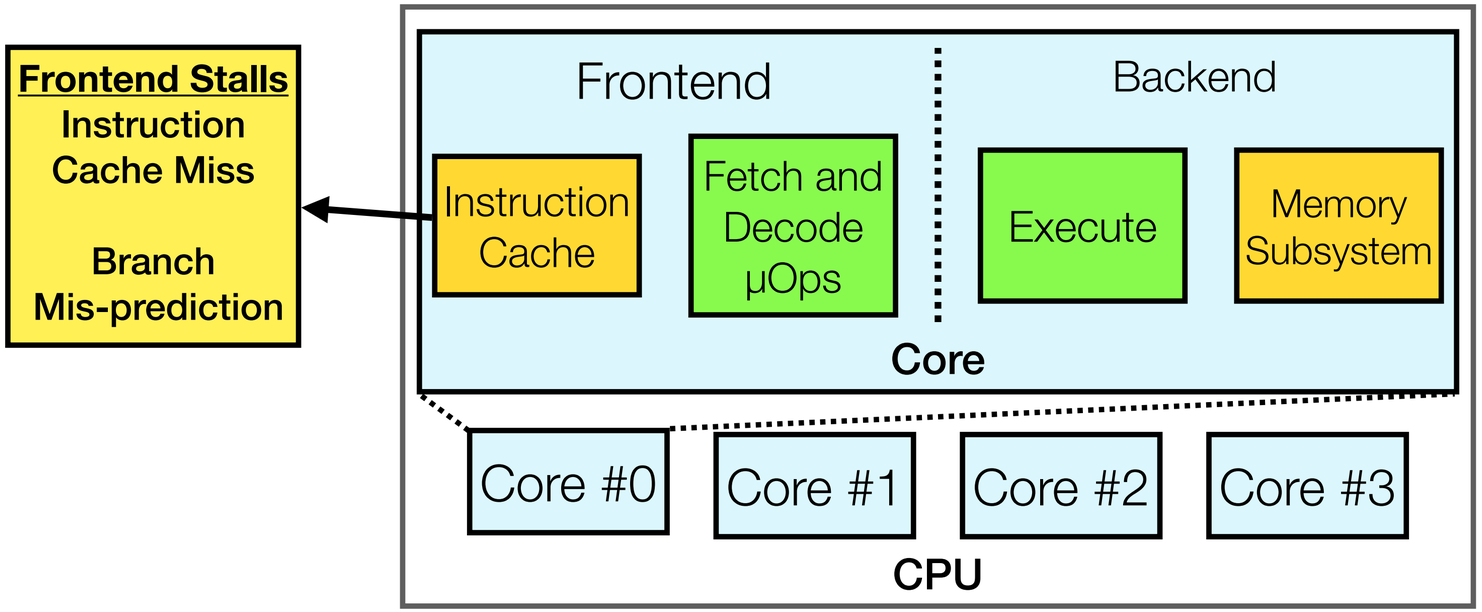
Frontend delivers up to 4 µOps per cycle to backend (Intel)

Micro-architecture Analysis

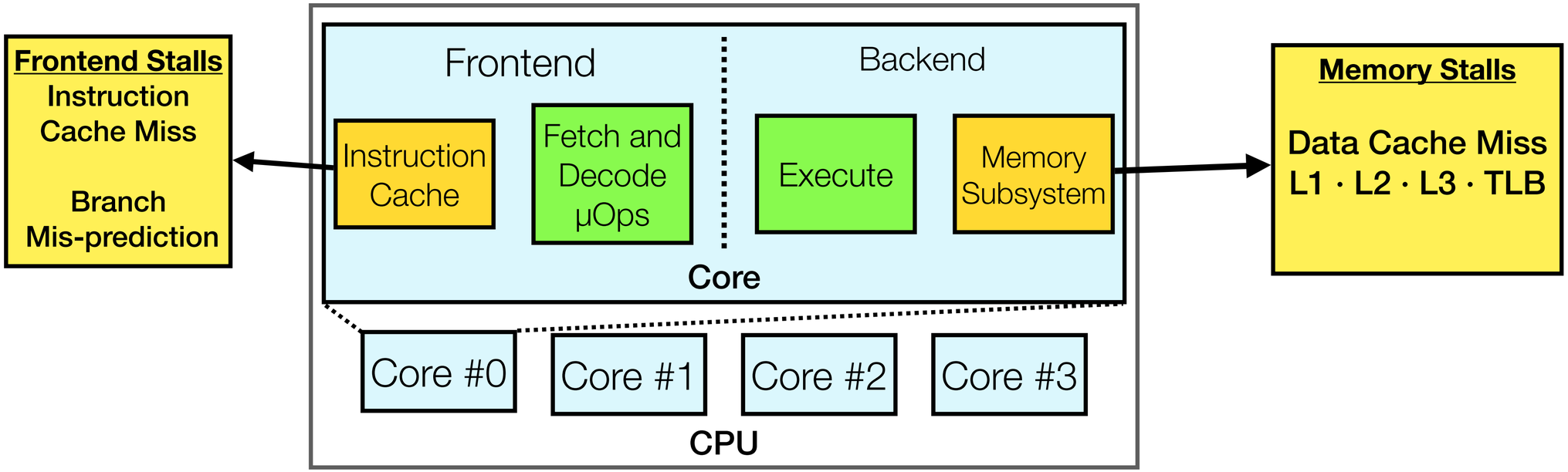


Provides data to registers from L1d, L2, LLC, and Main-Memory

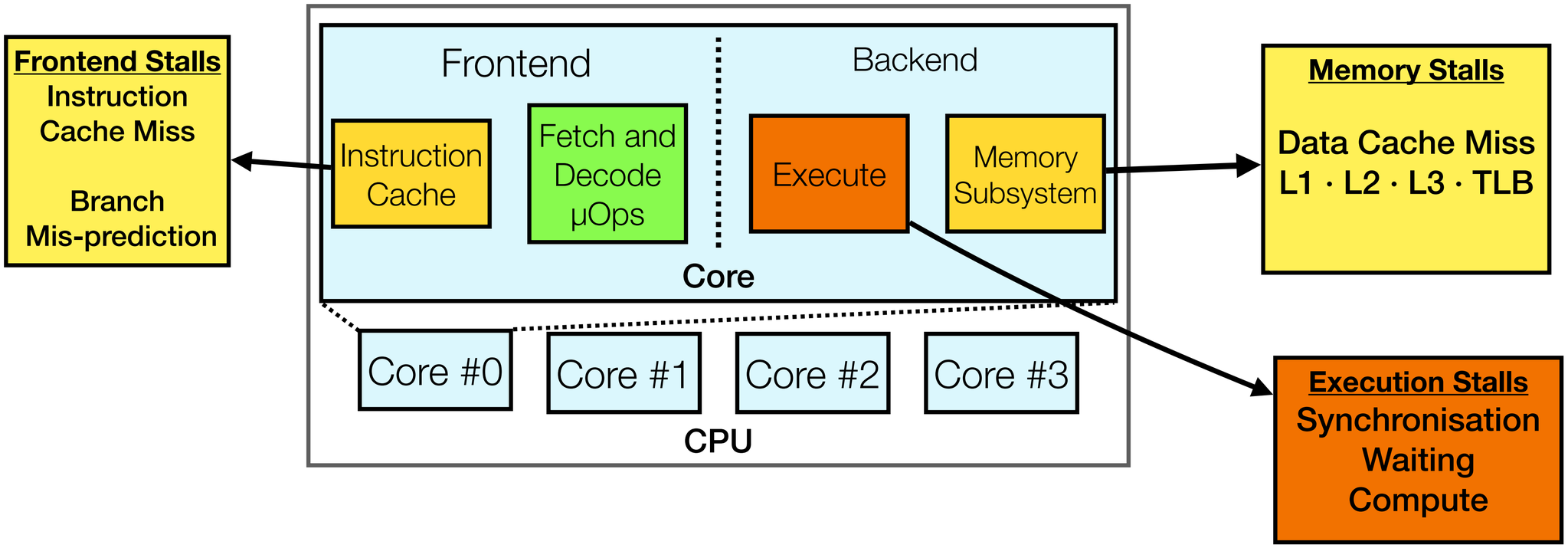
Micro-architecture Analysis



Micro-architecture Analysis

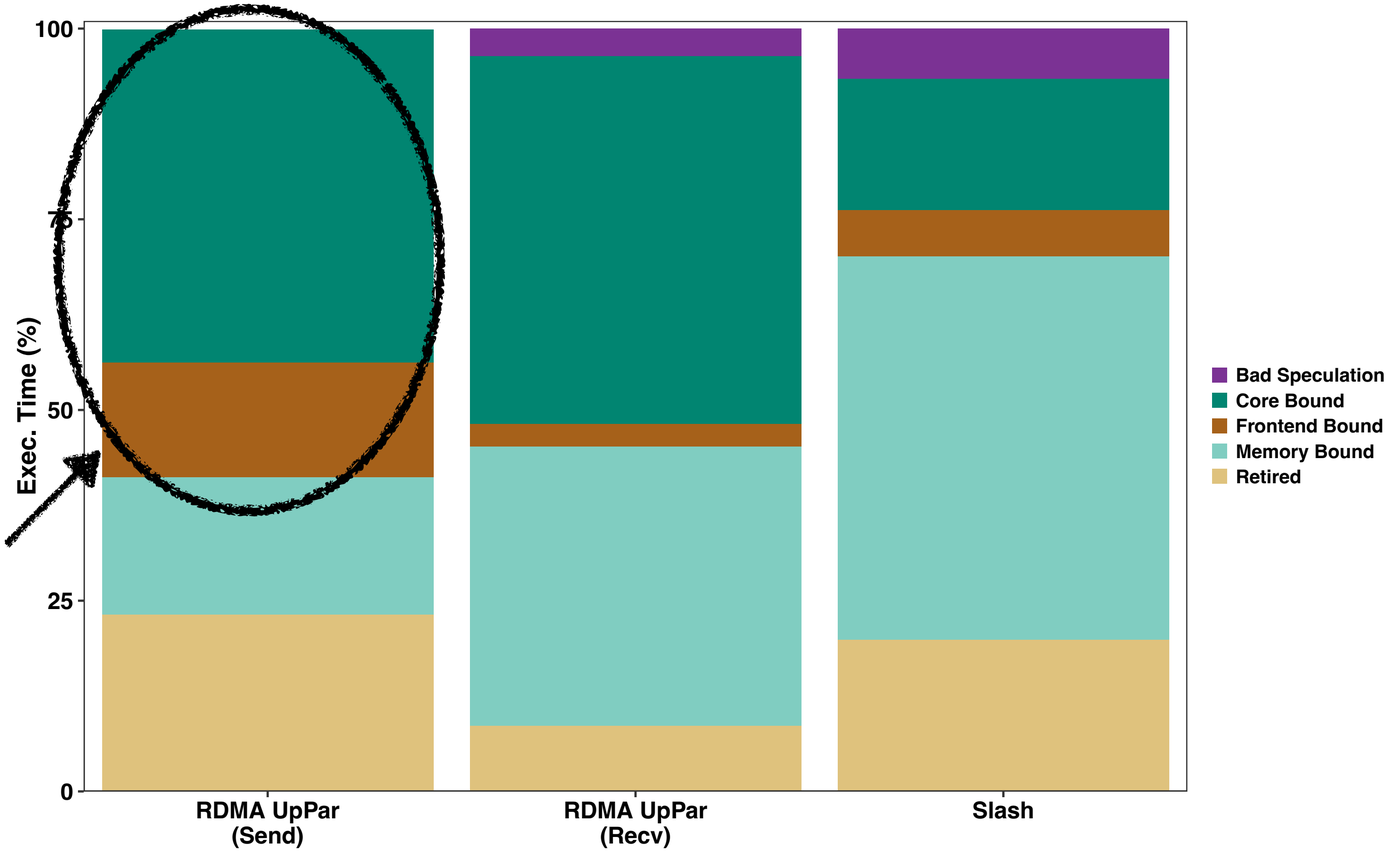


Micro-architecture Analysis



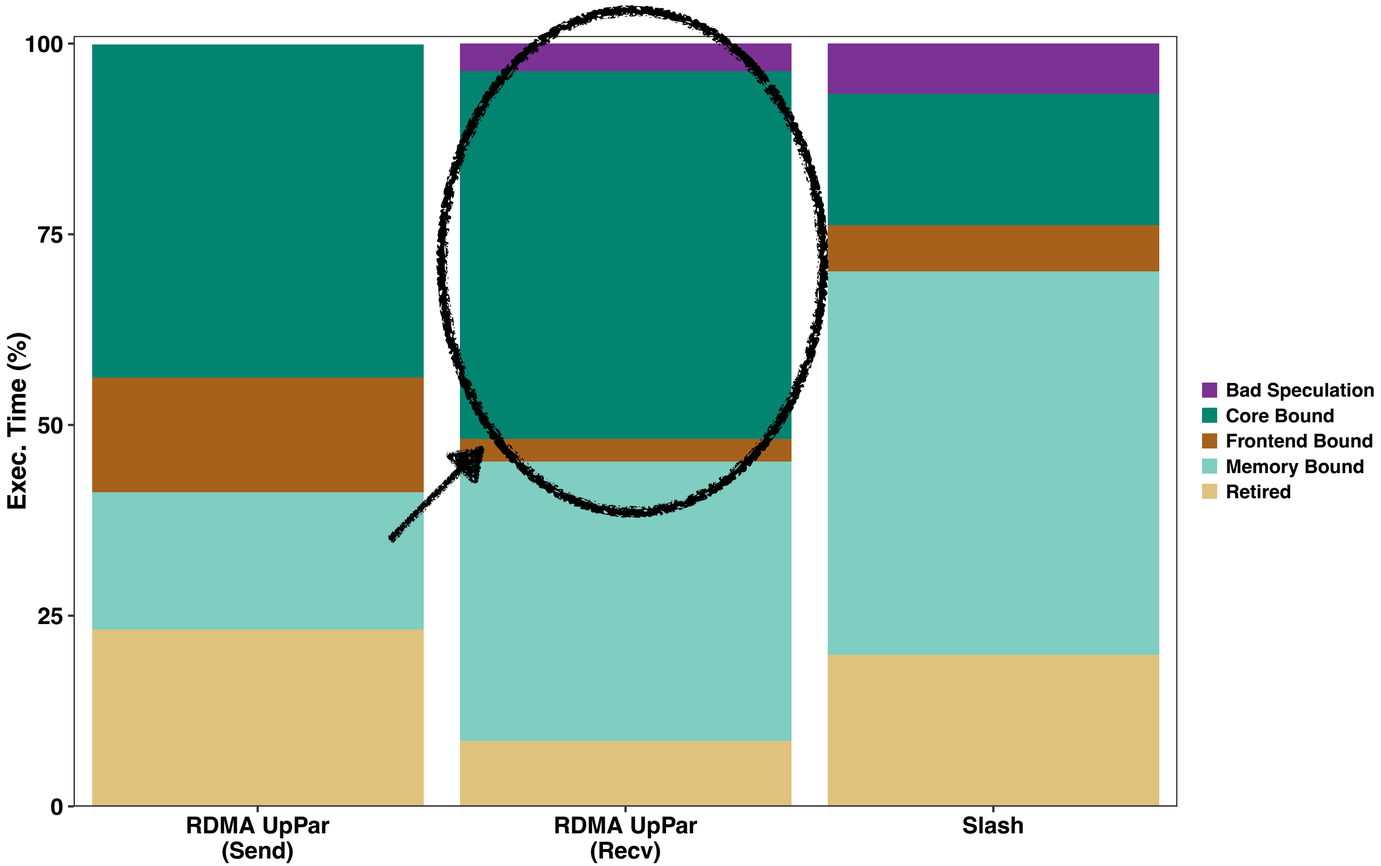
Hardware Performance Counters help us understand CPU performance

Slash Performance Gain Explained



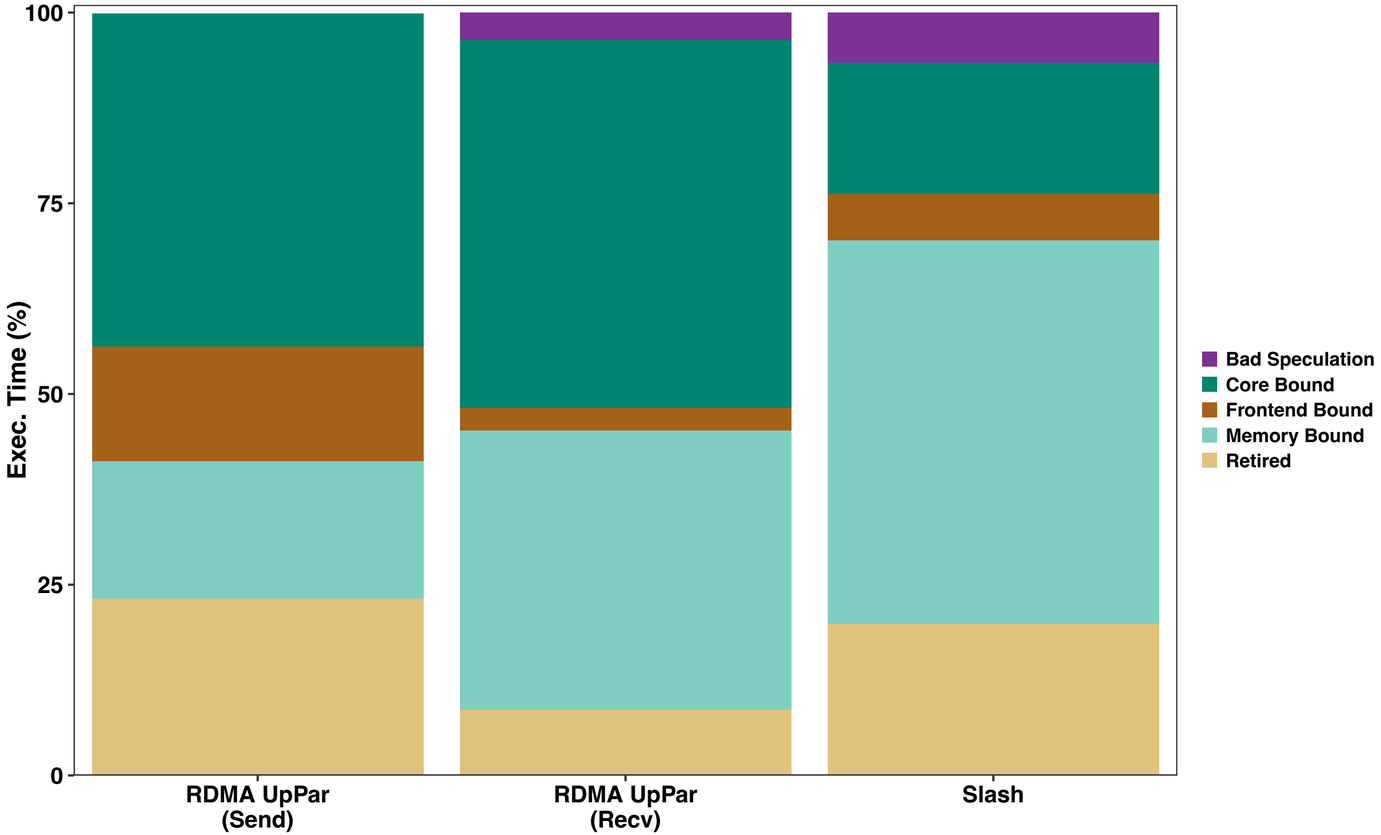
Sender of RDMA UpPar is Frontend and Core Bound
Partitioning involves complex code and spin waiting

Slash Performance Gain Explained



Receiver of RDMA UpPar spin waits on data from the sender

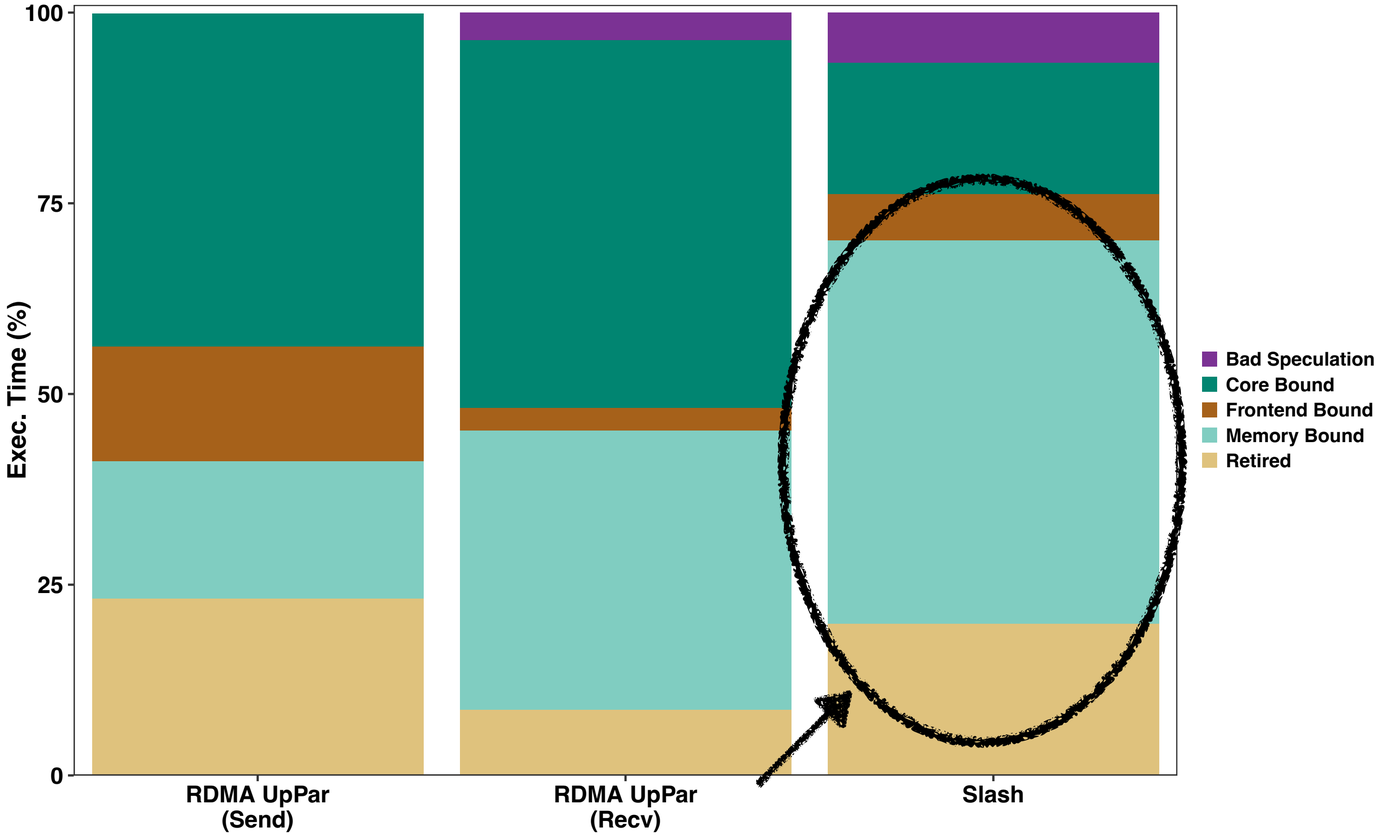
Slash Performance Gain Explained



RDMA UpPar limited by partitioning speed (CPU-Bound)

Receiver of RDMA UpPar spin waits on data from the sender

Slash Performance Gain Explained



RDMA UpPar limited by partitioning speed (CPU-Bound)

Slash is bound by memory speed

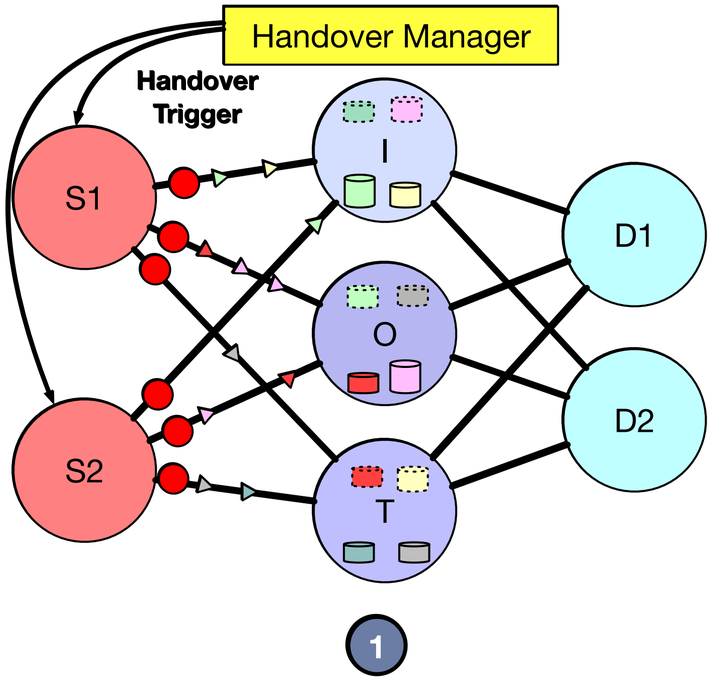
	IPC	Instr./ Rec.	Cyc./ Rec.
RDMA UpPar	0.6	166	274
	0.4	78	276
Slash	0.9	42	53

Slash waits for data to be materialized for processing

Backup

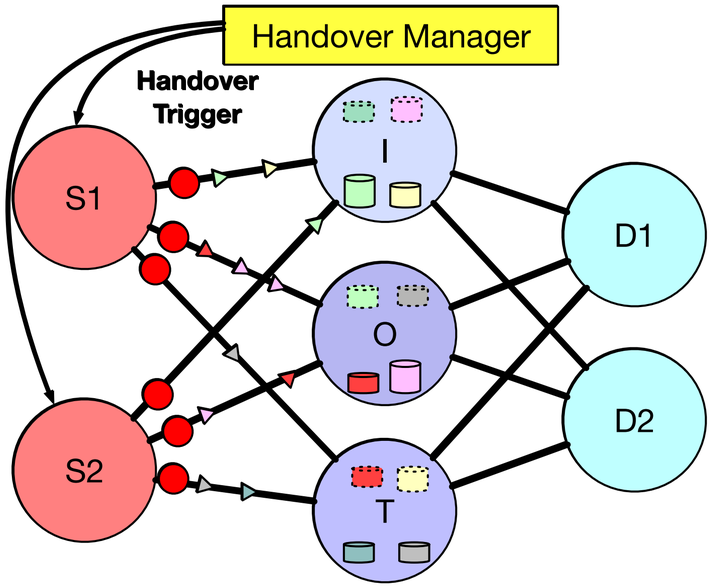
Rhino

Our solution: Rhino

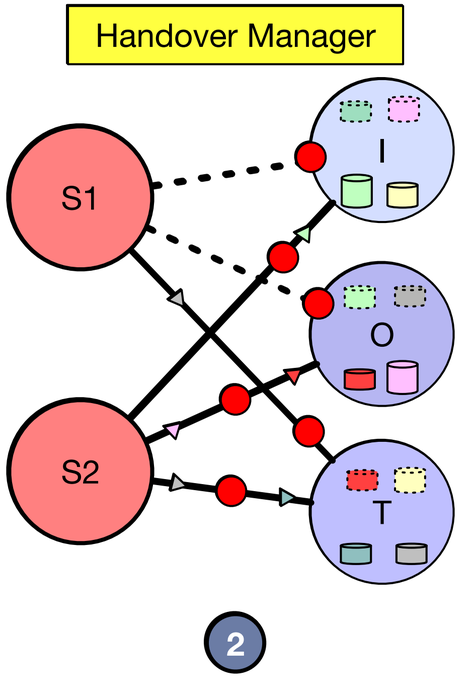


Handover markers to reconfigure instances O and T (red state) flow in the dataflow along with records

Our solution: Rhino

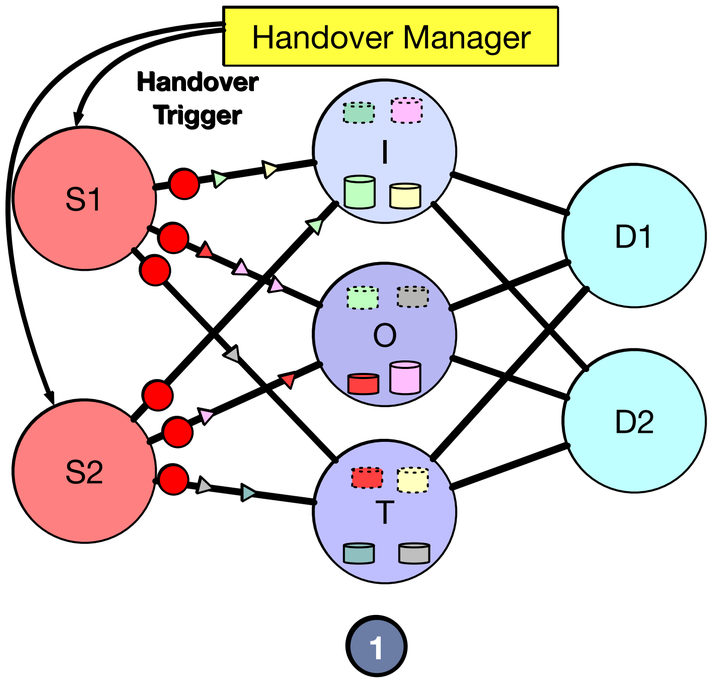


Handover markers to reconfigure instances O and T (red state) flow in the dataflow along with records



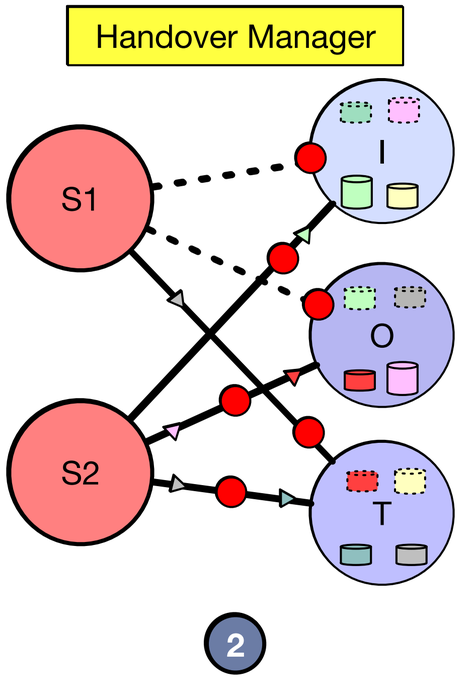
Handover markers reach operator instances

Our solution: Rhino



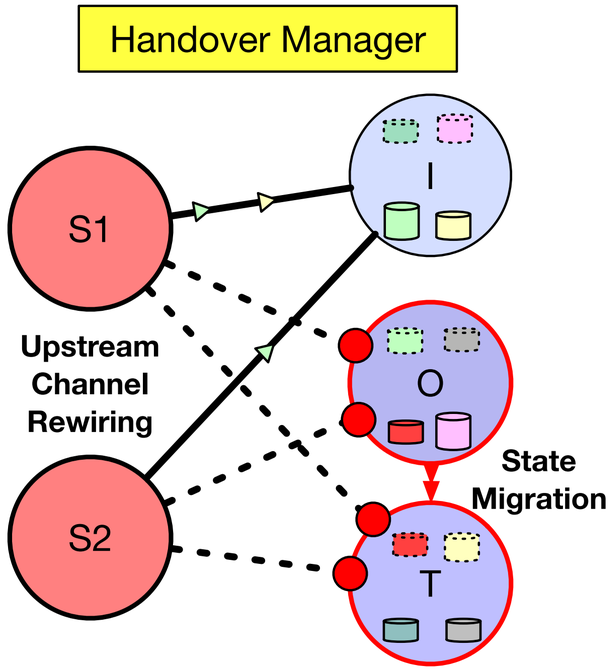
1

Handover markers to reconfigure instances O and T (red state) flow in the dataflow along with records



2

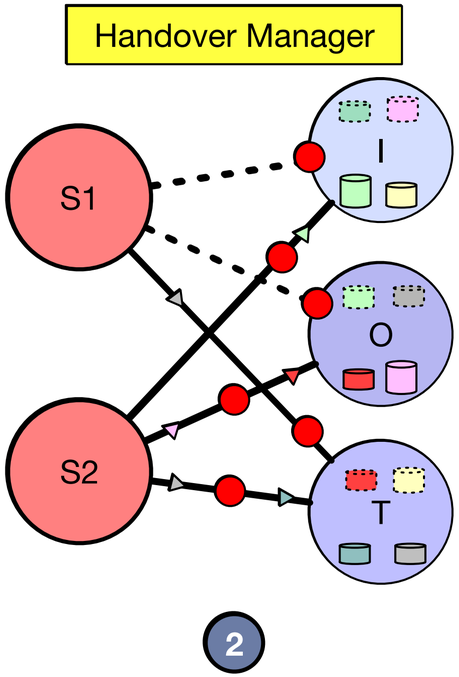
Handover markers reach operator instances



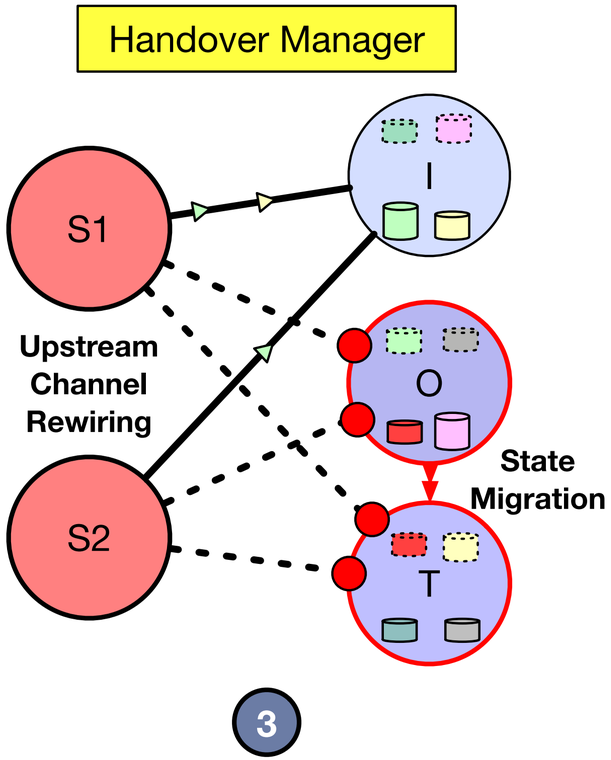
3

State and Task Handover between Origin to Target

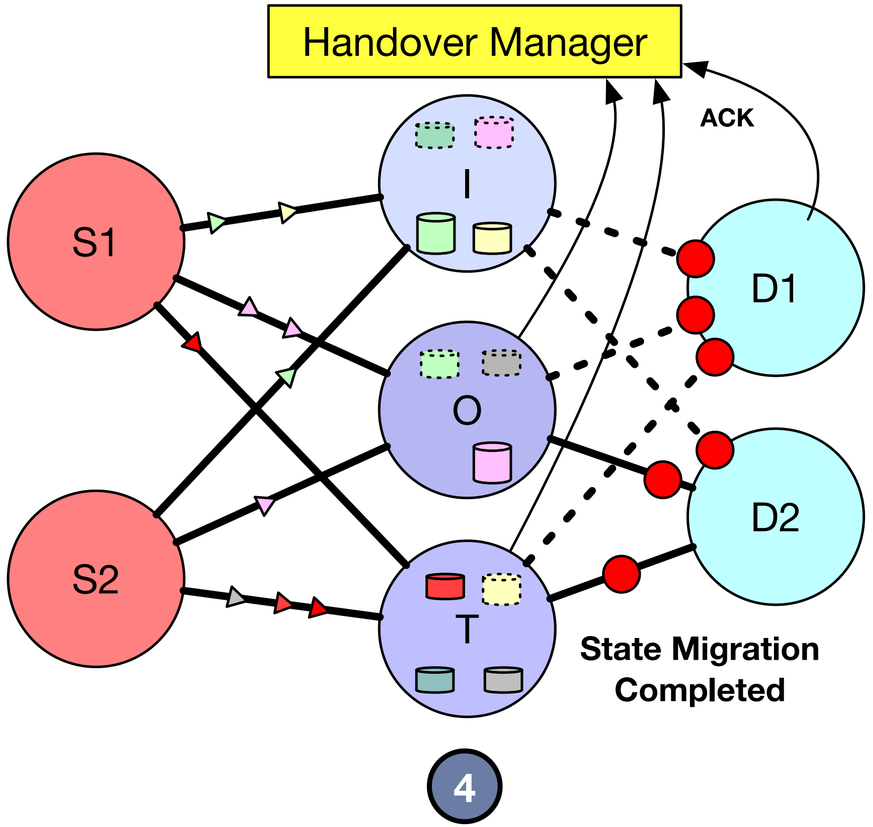
Our solution: Rhino



Handover markers reach operator instances



State and Task Handover between Origin to Target

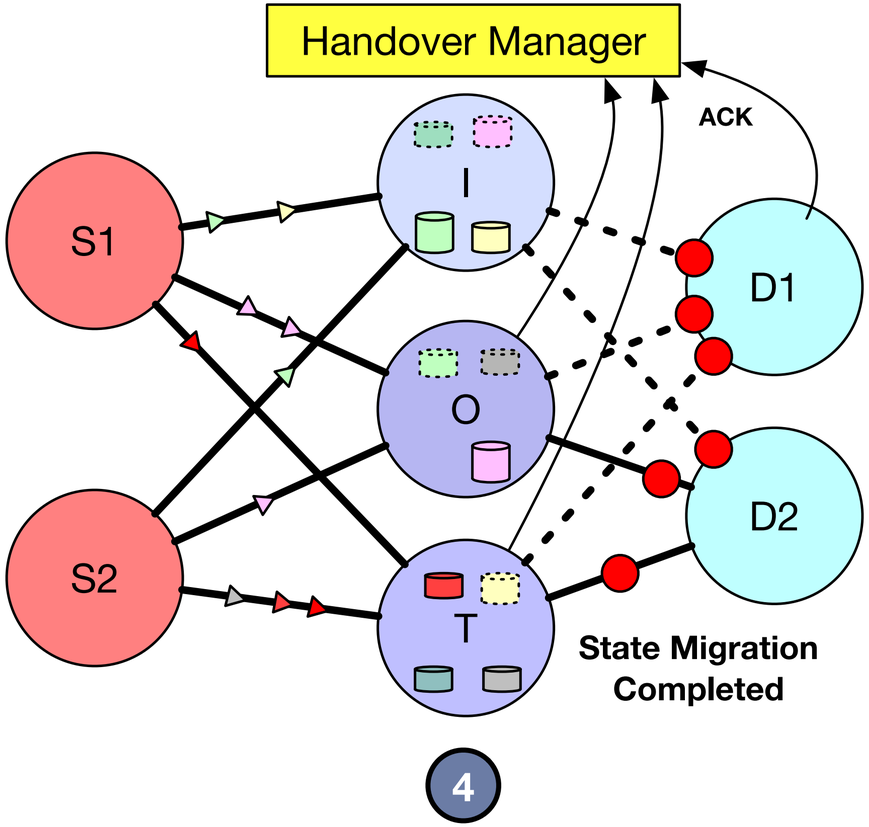


Operator instances forward handover marker and acknowledge reconfiguration completion

Our solution: Rhino

Handover Protocol to reconfigure running stateful query without halting it

Correctness based on dataflow properties: happened-before relation between markers and records



Operator instances forward handover marker and acknowledge reconfiguration completion

When to use Rhino

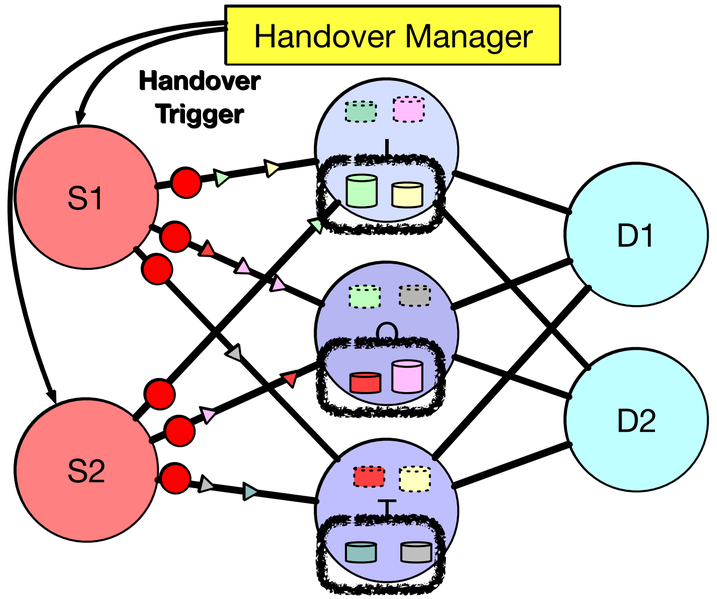
- Cost of restarting query violates SLO
- Cost of proactive state migration is still affordable (compared to original reconfiguration mechanism of target SPE)

Rhino+Spark

- Trigger handover at micro-batch (RDD) boundaries
- Finer granularity: trigger handover at stage-boundaries
- State Migration:
  - if state is RDD: replicate RDD incrementally
  - if state is in LSM-Tree: take incremental snapshot and use state-centric replication

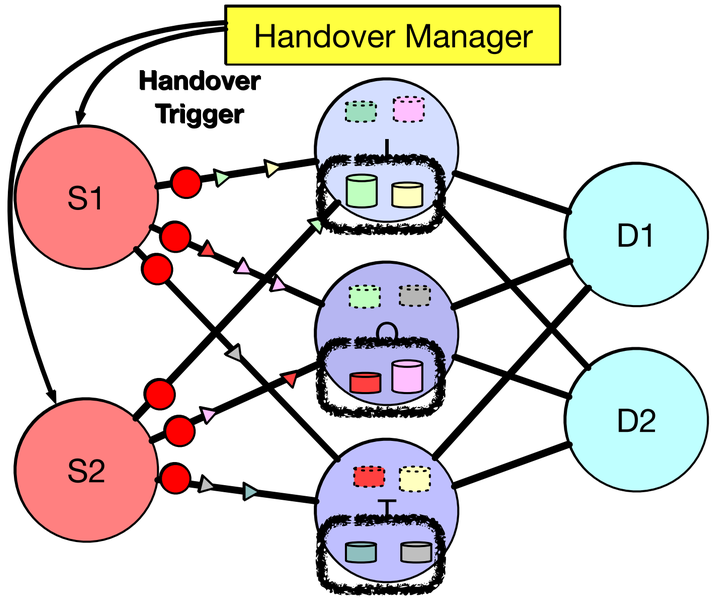
Consistent hashing with virtual nodes

- Split state of each operator instance into logical groups based on key
- Consistent hashing reduces (k,v) remapping after rehashing



Consistent hashing with virtual nodes

- Split state of each operator instance into logical groups based on key
- Without CH, remapping after rehashing involves potentially all keys
- CH reduces remapping after rehashing to k/m keys
- CH with virtual nodes remaps only the keys in a virtual node



Types of fault-tolerance for SPEs

- Transactional: MillWheel, each state update/produced record is a transaction
- Lineage: Spark Streaming, track and persistent input/output dependencies
- Checkpointing: Flink, variant of Chandy-Lamport snapshotting algorithm
- Change-log: KafkaStream, persist metadata changeling in commit-log

Rhino requirements on host system

- R1: Streaming dataflow paradigm: tuple-at-a-time or BSP
- R2: Consistent hashing with virtual nodes
- R3: Mutable state, need to R/W state

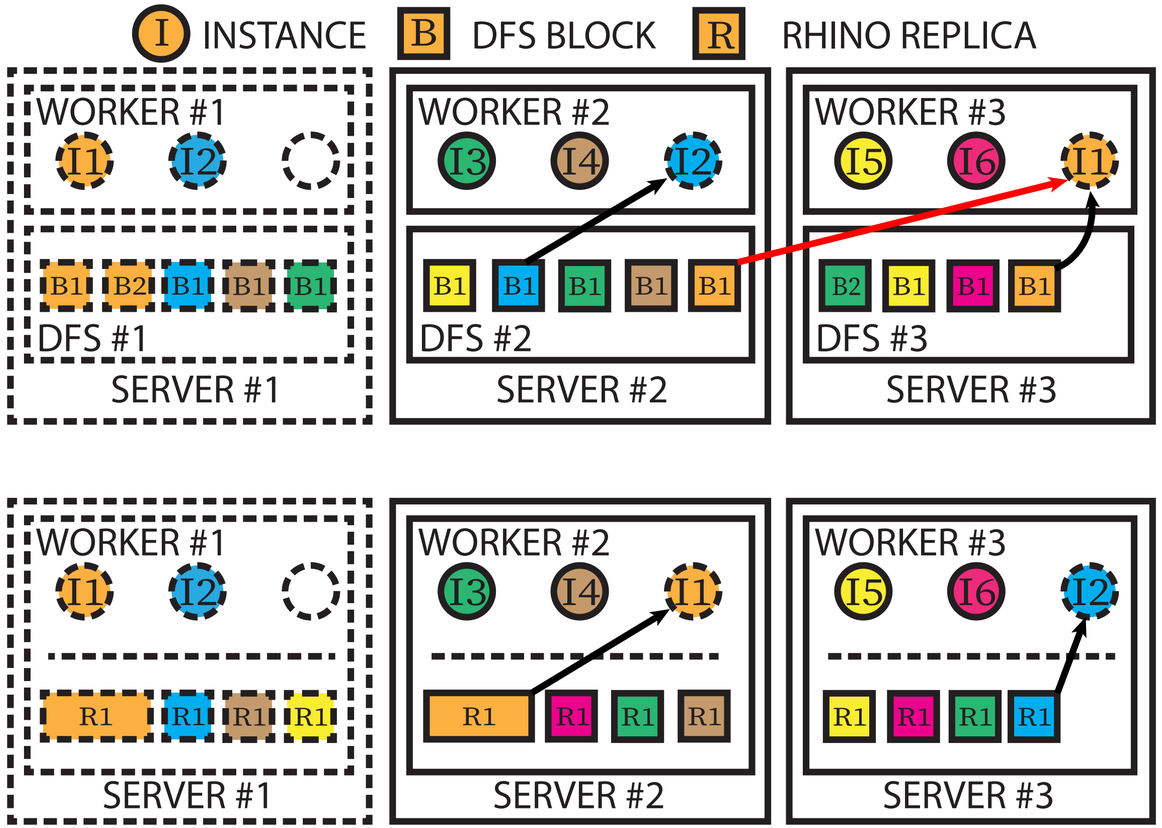
Rhino or Megaphone

- State migration in Megaphone is reactive and programmable in DSL
  - Megaphone uses a migration operator in the dataflow program
- State migration is proactive to serve further reconfiguration transparently to end-user
  - Rhino pipelines checkpointing and migration
- Rhino has in-band synchronisation: markers flowing alongside data records
- Megaphone uses out-of-band synchronisation: only TimelyDataflow/MillWheel but costly on SPEs that rely on in-band synchronisation

Block-based replication is not enough

Block-centric

State-centric



Pipelined State Snapshots for SPEs

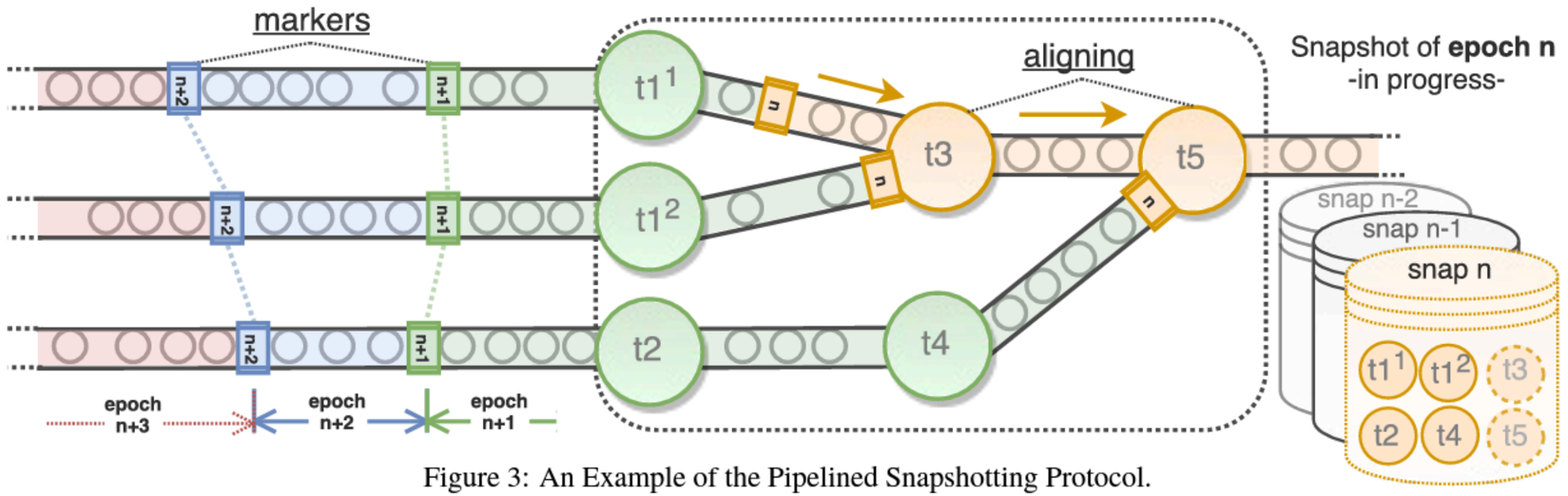
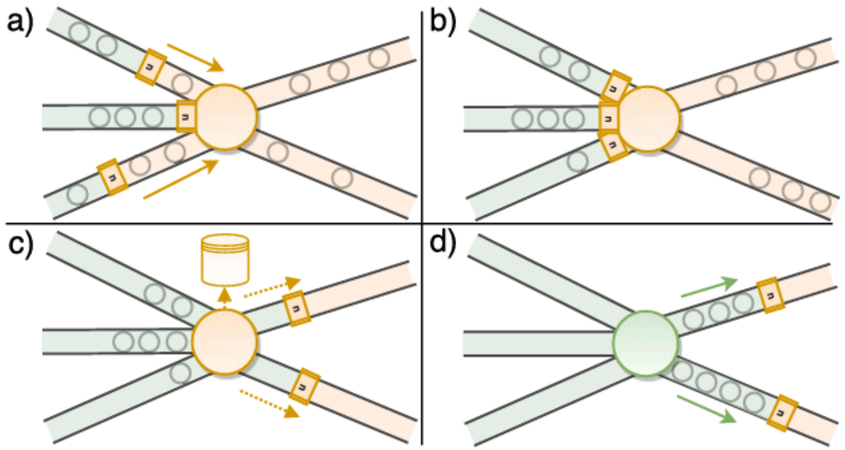
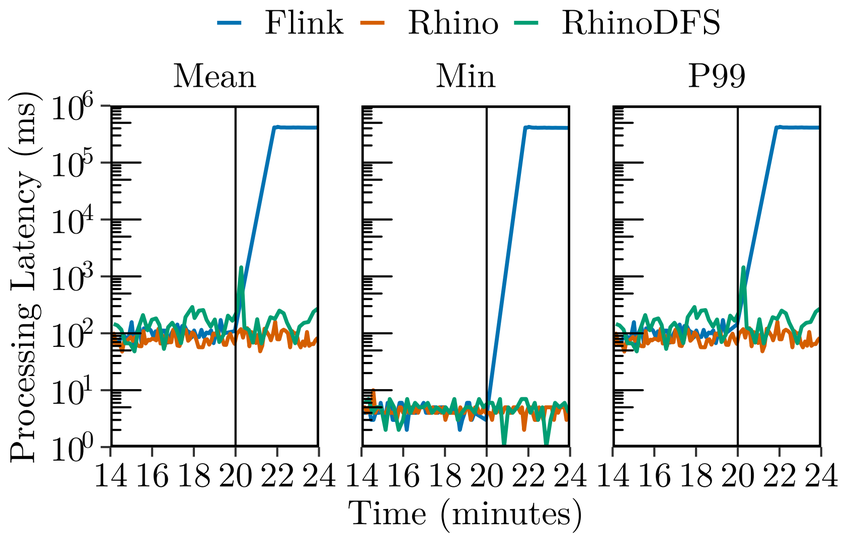


Figure 3: An Example of the Pipelined Snapshotting Protocol.

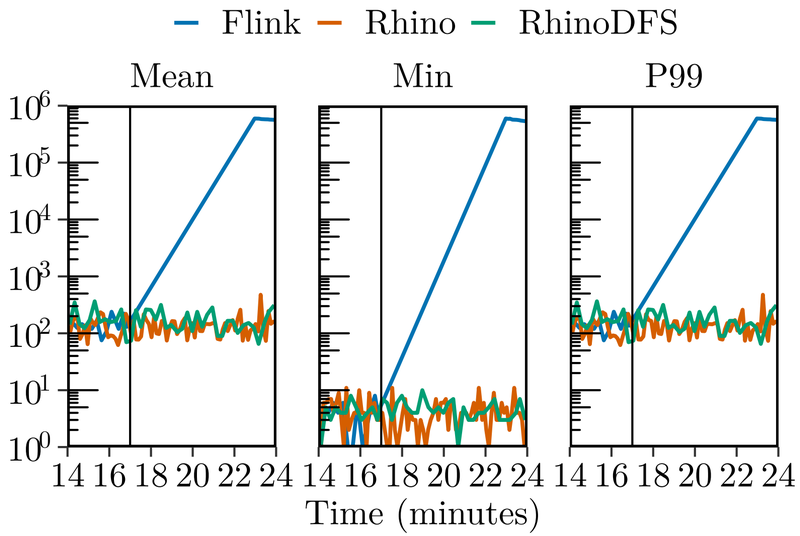


Source: Carbone et al., State Management in Apache Flink, VLDB'17

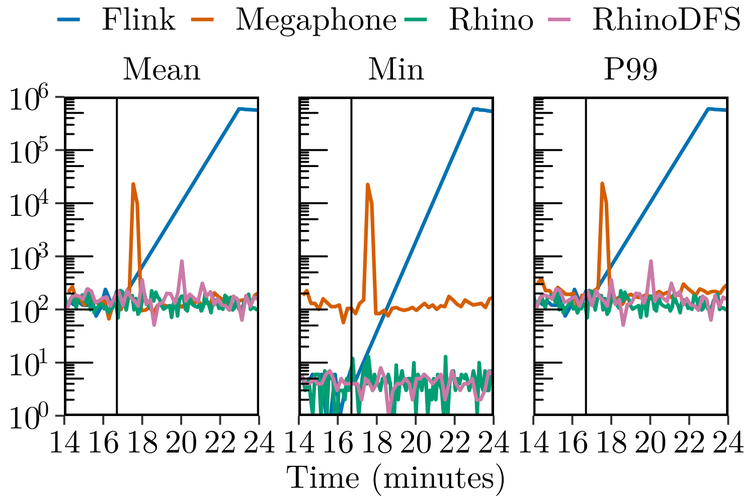
End-to-end Evaluation (NBQ8)



Fault-tolerance



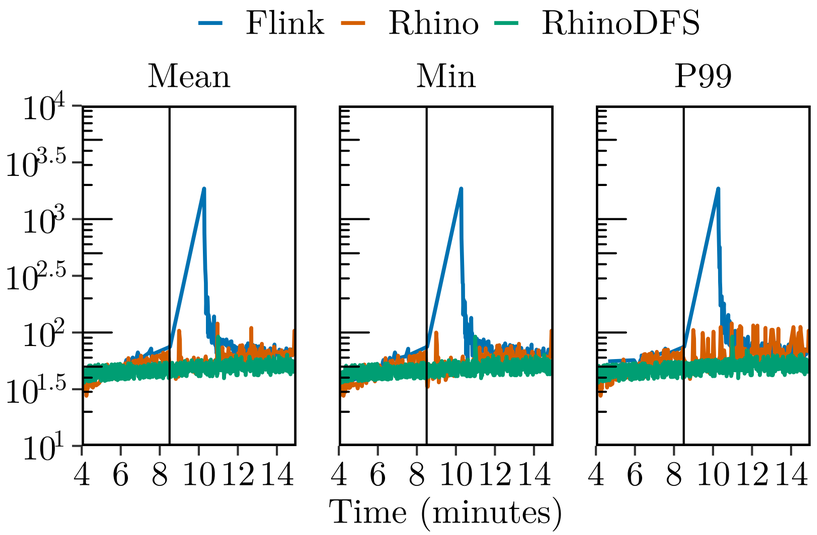
Vertical Scaling



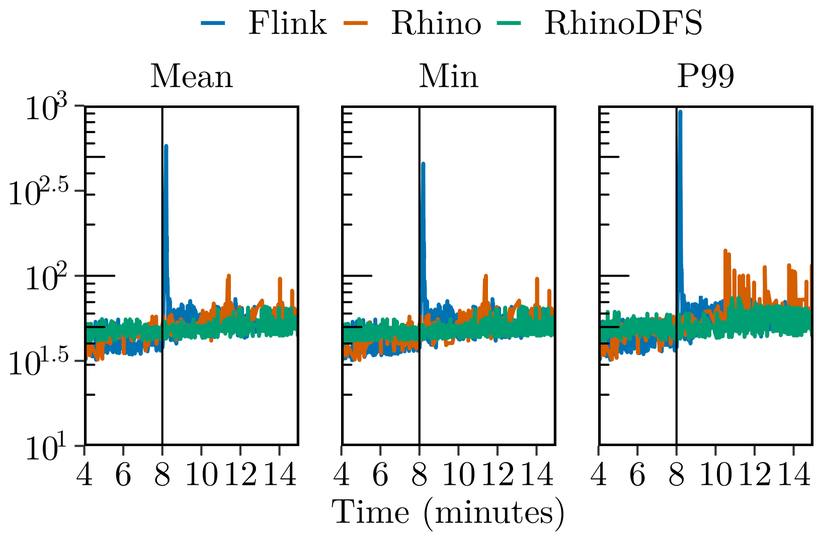
Load Balancing

State Size: ~190 GB

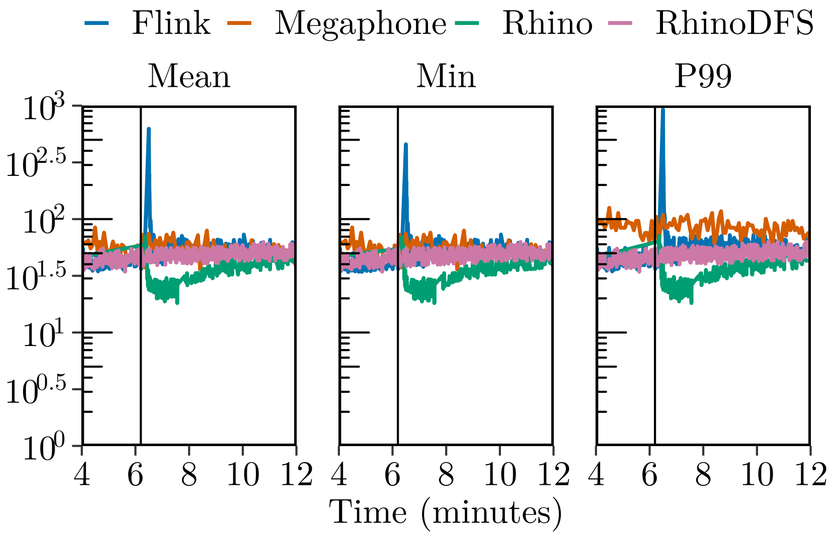
End-to-end Evaluation (NBQ5)



Fault-tolerance



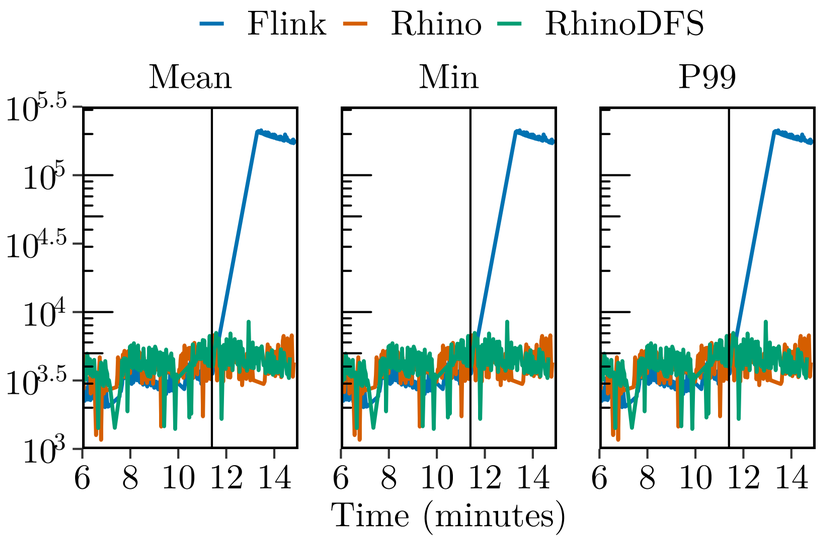
Vertical Scaling



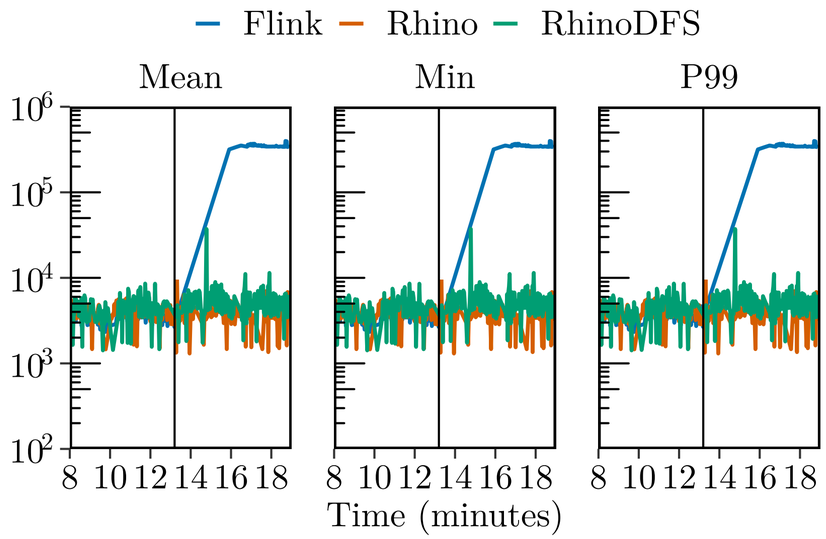
Load Balancing

State Size: ~26 MB

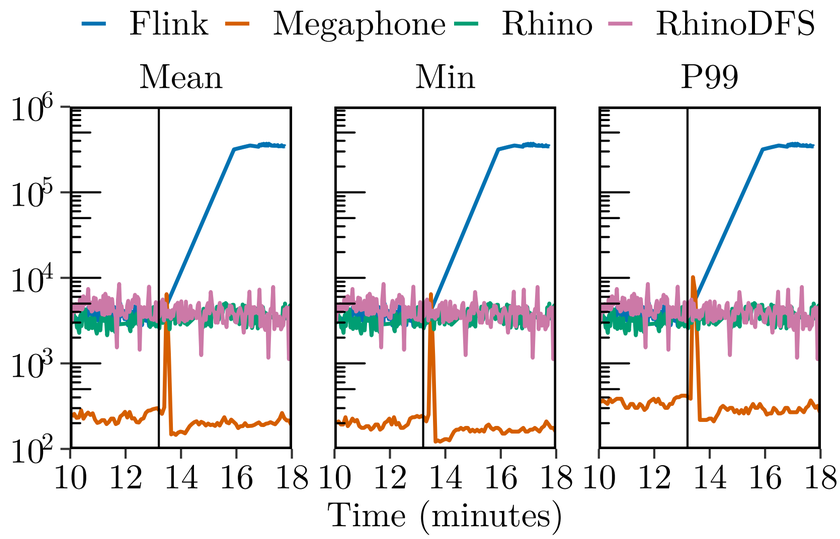
End-to-end Evaluation (NBQX)



Fault-tolerance



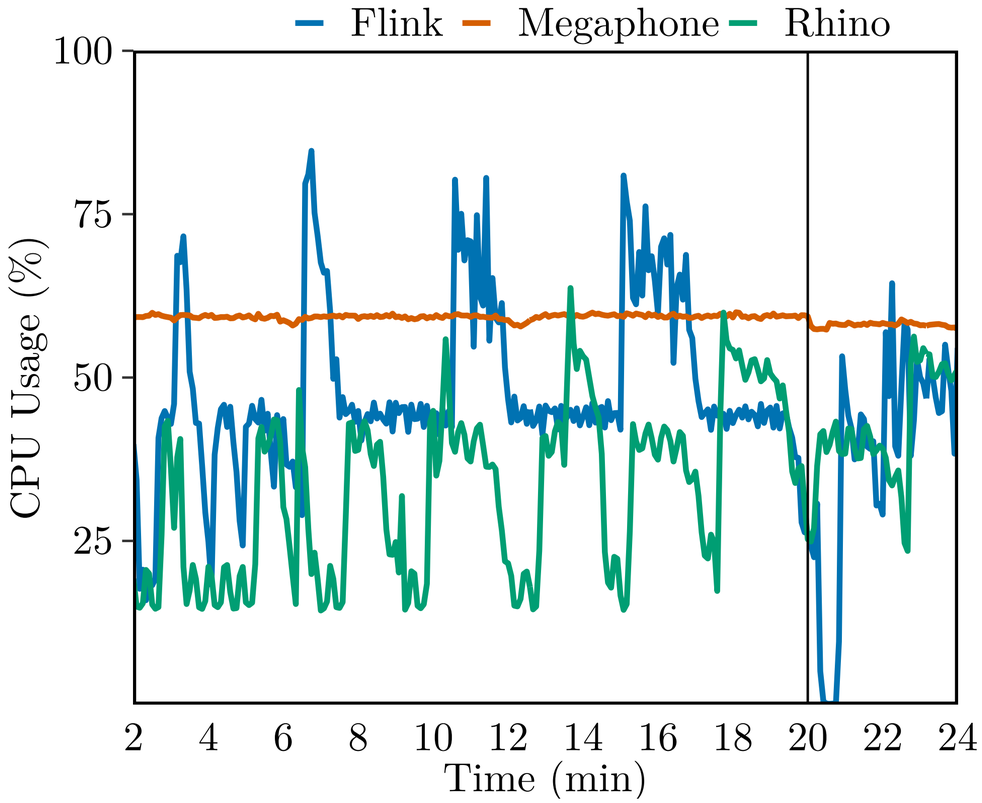
Vertical Scaling

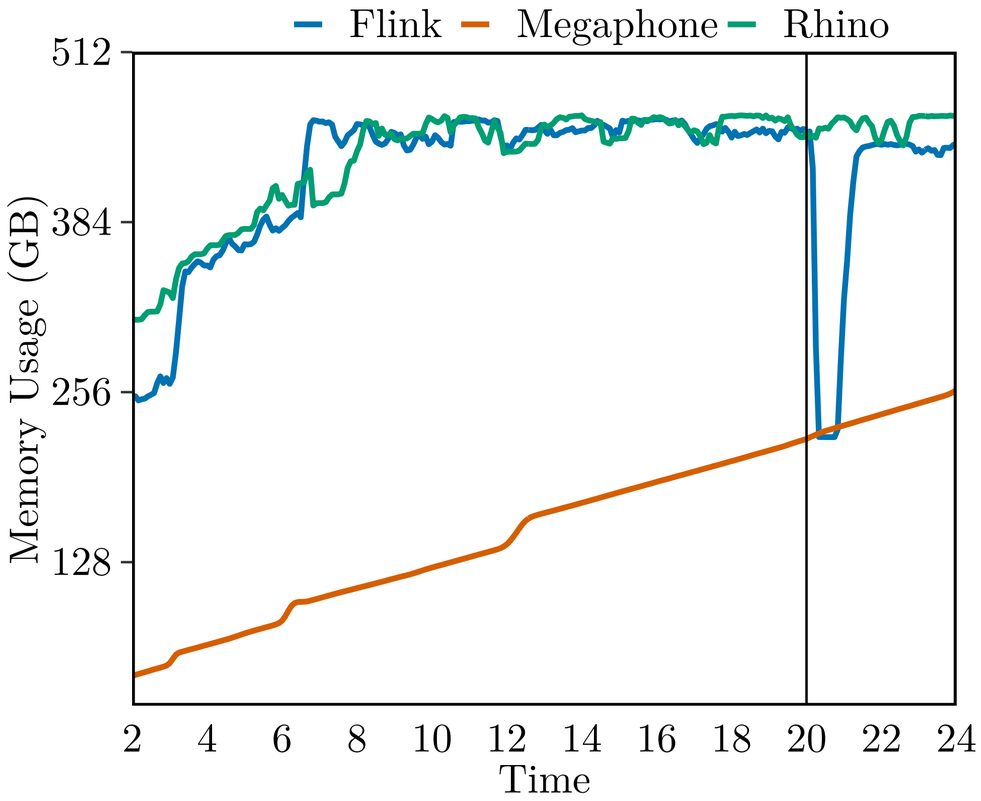


Load Balancing

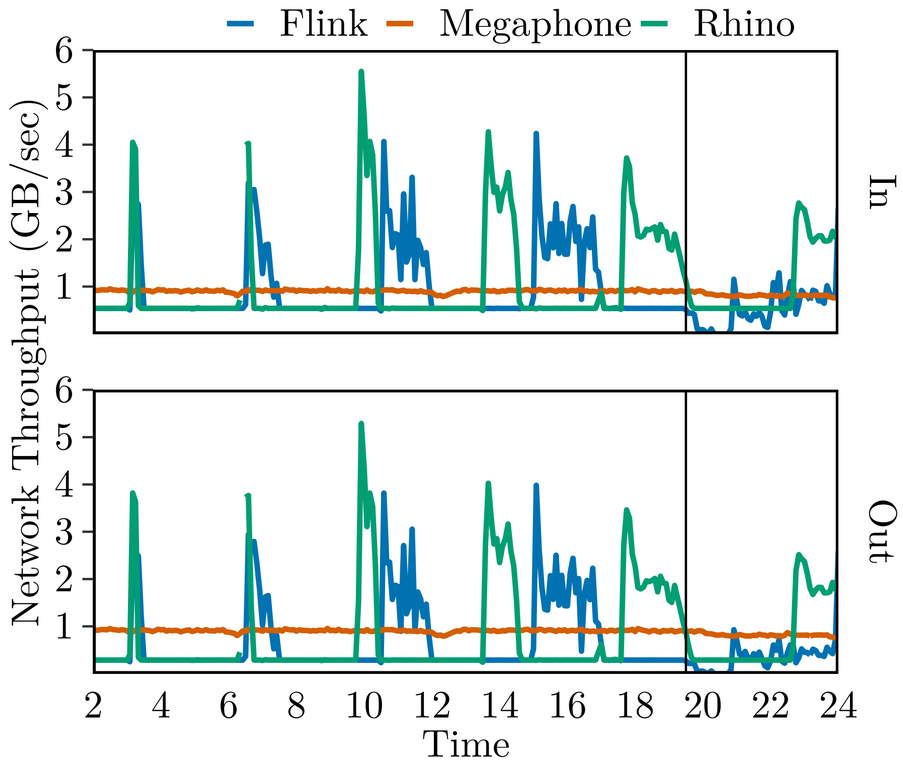
State Size: ~180 GB

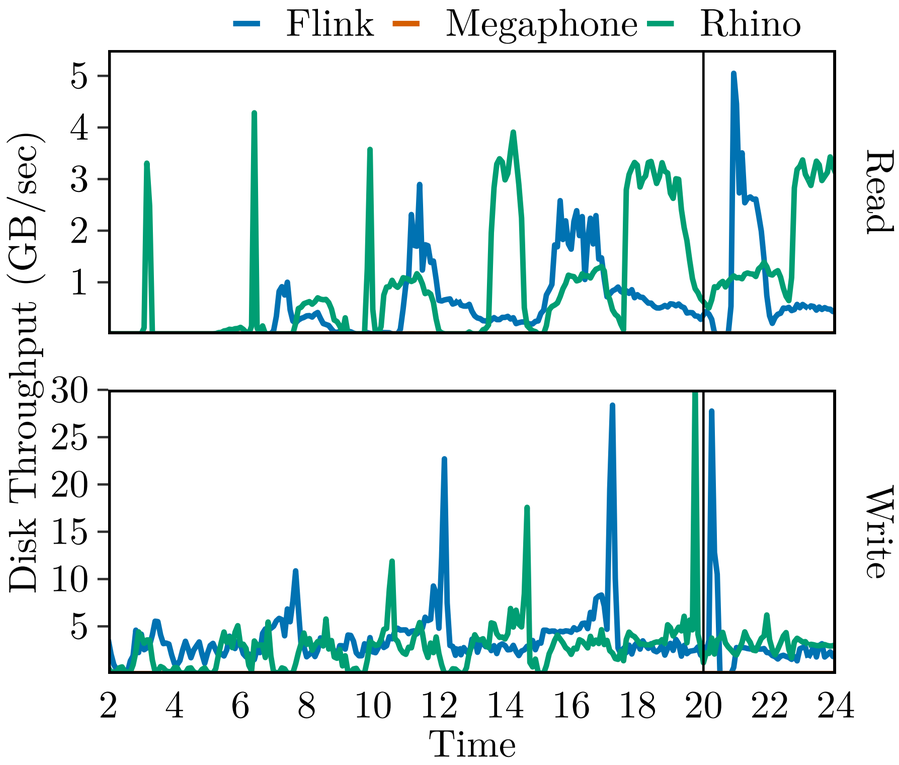
Resource Utilisation (NBQ8)



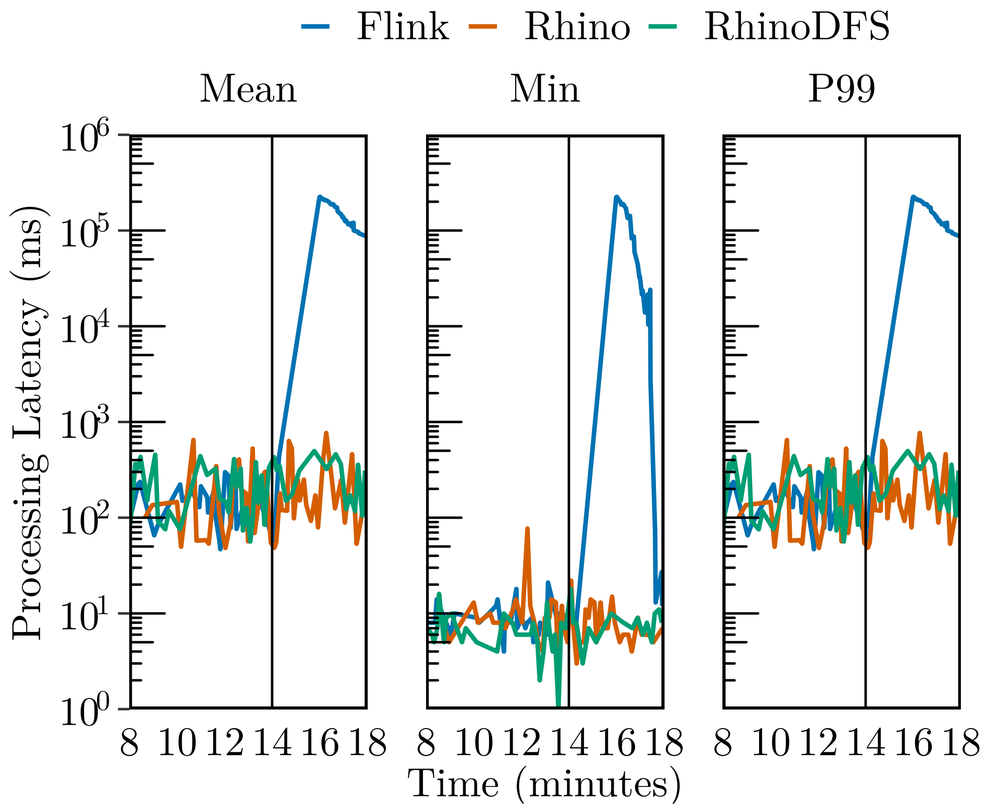


Resource Utilisation (NBQ8)





Fluctuating Data Rate



Rhino correctness

Theorem 1 *Consider a handover that migrates the state S^{t-1} of the virtual node $(k_l, k_q]$ from O to T at timestamp t. The protocol guarantees that: 1) T receives S^{t-1} at t and then processes records with keys in $(k_l, k_q]$ and timestamps greater than t and 2) the handover completes in finite time.*

Backup

Misc

Sizing SPE resources

- Consider Source->windowByKey->Sink
- Input: record format, message/sec, window length, (k,v)-pairs format, num keys
- Ingestion bandwidth: records size * messages/sec
  - How many servers for Source? Network throughput (per server)?
- Shuffling bandwidth: ingestion bw / num of consumers
  - Memory shuffling bandwidth M for l local consumer(s)
  - Network shuffling bandwidth N for r remote consumers
  - Determine state write speed on each consumer

Sizing SPE resources

- On each consumer we have state size = num distinct keys * (k,v) size
- Determine output speed based on state size
- Based on the above, determine number of servers to handle window operator and sink
- Add checkpointing?