Resource Efficient Stream Processing Platform with Latency-Aware Scheduling Algorithms

Yuta Morisawa,　Masaki Suzuki,　Takeshi Kitahara
KDDI Research, Inc.

Background

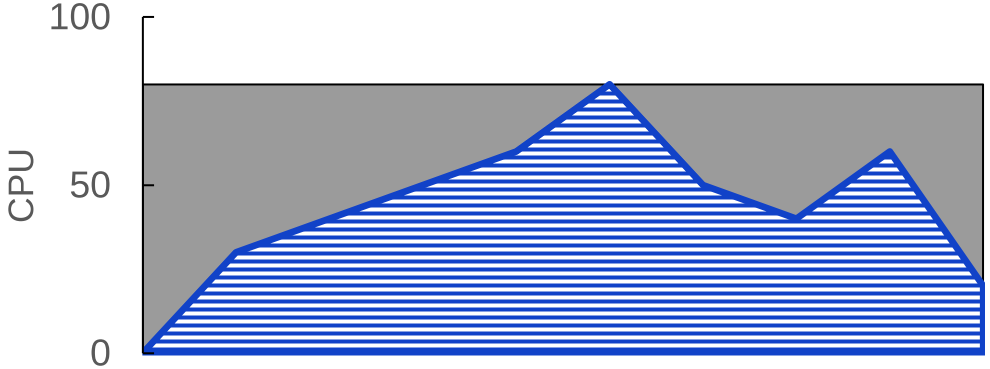
■ Stream processing has become a major region

- By 2025, nearly 30 percent of the data will be real-time [1]
- One organization will have many stream applications for their business

■ Resource Inefficiency of the traditional big data stream processing

- Users must allocate resources according to traffic peak
- Once users creates a cluster, the total amount of resources (e.g., CPU cores) cannot be modified

Goal: Improvement of resource efficiency in an environment where multiple applications with different latency requirements exist



Timeline of the CPU usage. Gray area shows unused CPU resources.

[1] David Reinsel – John Gantz – John Rydning, "The Digitization of the World From Edge to Core", Nov. 2018,

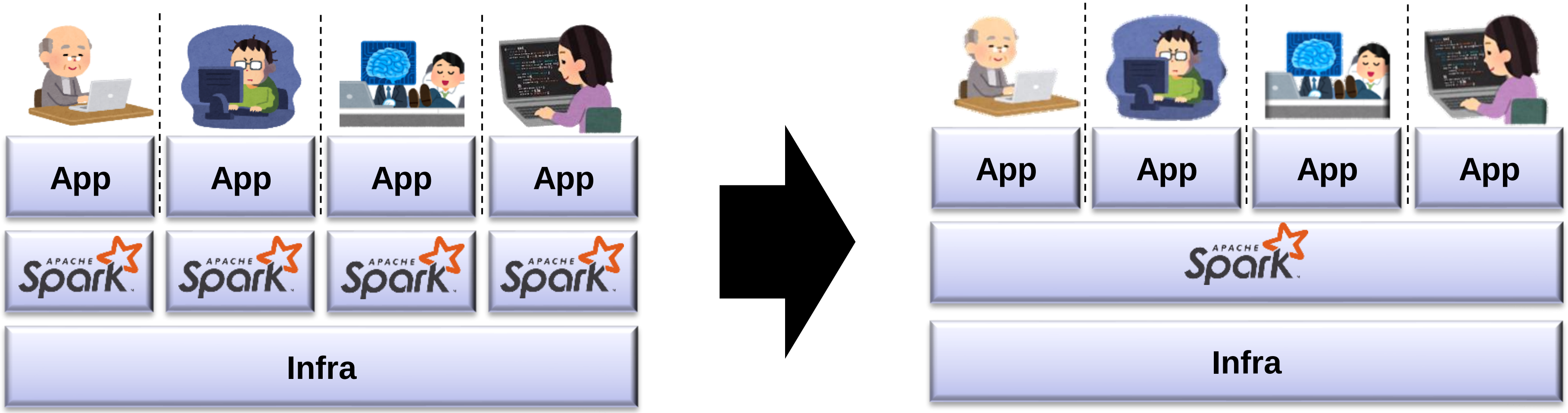
Challenges

Improving resource efficiency of the stream platform is difficult because

- **Different latency requirements**
  - Stream applications have various latency SLAs that must be observed
- **Unexpected traffic patterns**
  - Input data rate often varies suddenly
  - Sometimes, users must allocate extra resources to cope with it
- **Optimization for multi-applications environment**
  - Improving the resource efficiency of every cluster is needed to minimize the platform cost
  - Prior studies did not address the problem

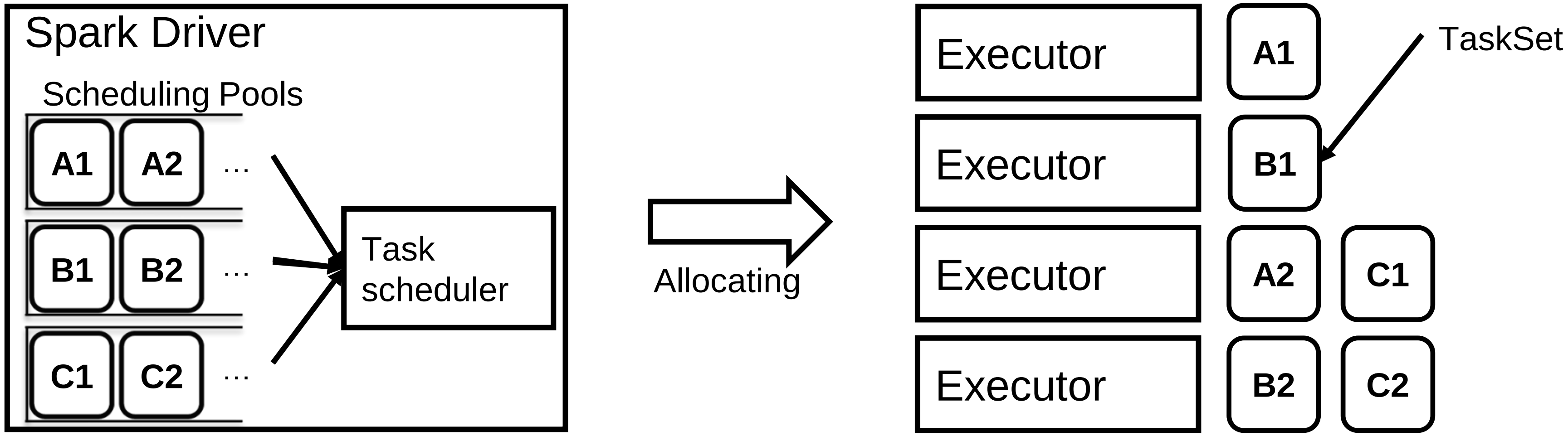
Proposal: Multi-application platform

- **Accommodate multiple applications in one Spark cluster**
  - This design enables finest granularity resource management
  - Resource reallocation can be completed with lower latency
- **Latency-aware task scheduler**
  - Determine the priority of each application to observe applications' latency SLAs



Proposal: Architecture Overview

- **Scheduling pools**
  - Applications are assigned to dedicated scheduling pools
  - Scheduled in the task level granularity
- **Executors**
  - Just runs assigned tasks (do not care what applications are)
- **API**
  - Fully compatible with Apache Spark



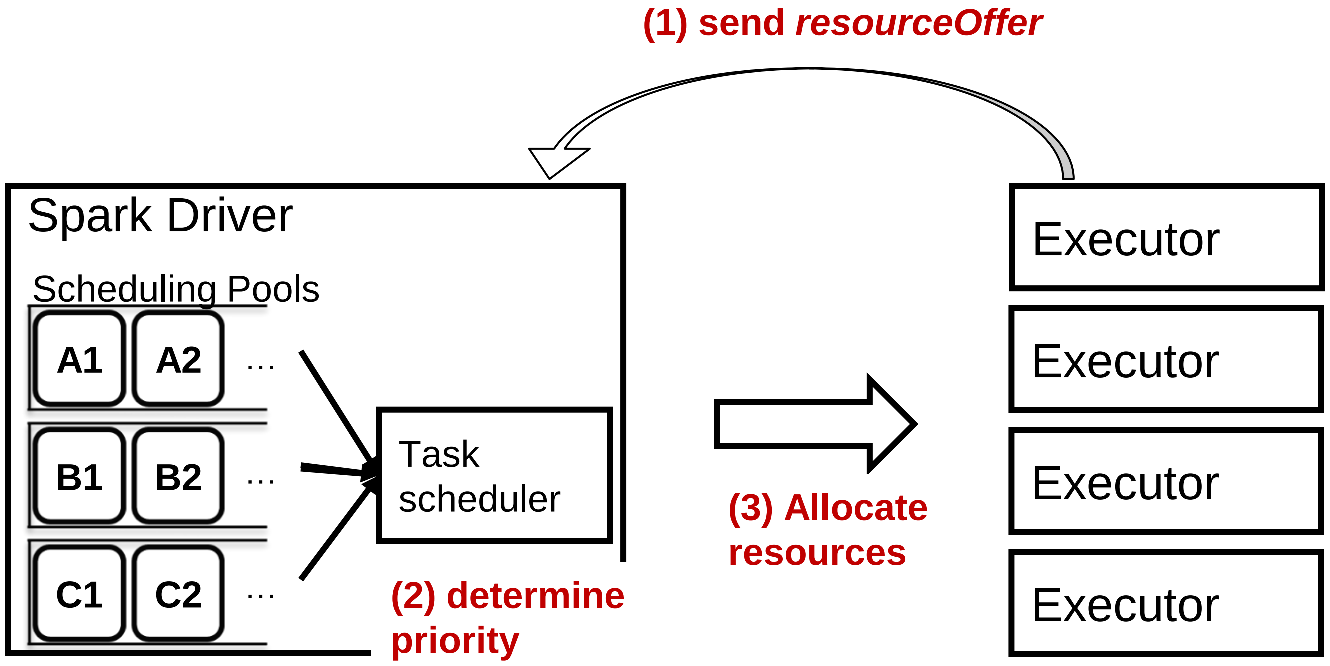
Proposal: Latency-aware task scheduler

Scheduler designs

- White box: Estimating the necessary resources
  - Difficult to apply to stream applications because data rate varies in time
- Black box: Using only metrics (we chose this design)
  - Monitor the latency and manipulate resource allocation

How Black box design works

- Utilized existing Spark task-level scheduler
  1, *resourceOffer* is issued when an Executor has available resources
  2, Task scheduler calculates priorities of the each task in each application
  3, The scheduler allocates resources according to the priorities



Proposal: Latency-aware task scheduler

P_s	Priority of Job_s
T_c	Current Time
T_s	Submission time of Job_s
L_s	Latency SLA of Job_s

■ Implemented 3 algorithms

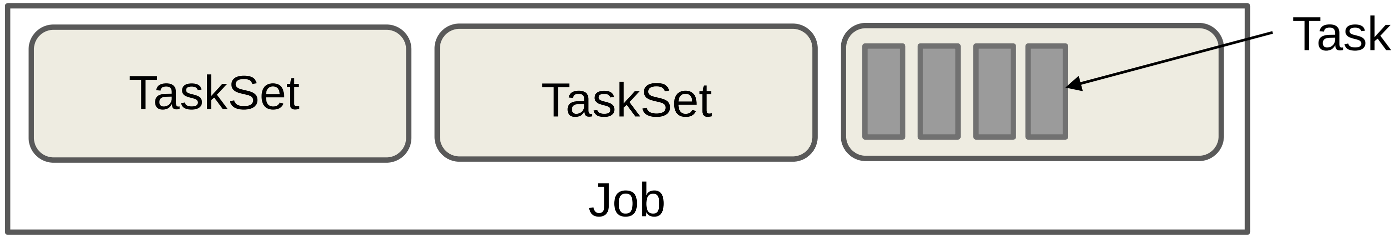
- Earliest Deadline First (EDF), Priority based EDF, Process time Estimation
- Goal: Determine the priority of the tasks of each application to observe SLAs

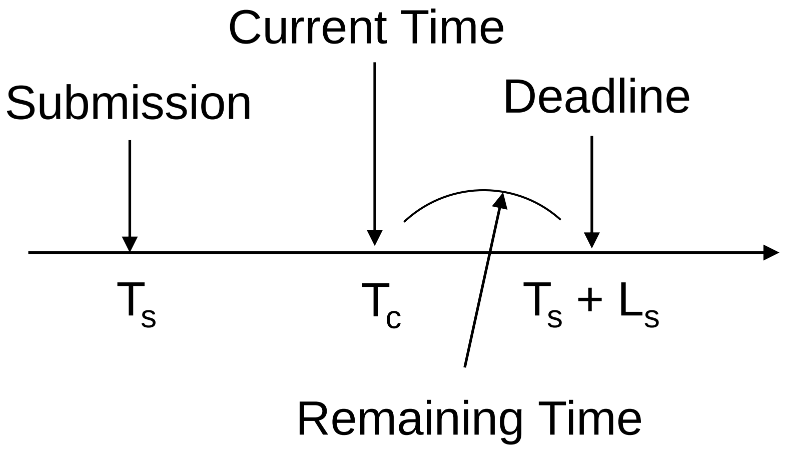
■ EDF

1, Scheduler calculates P_s for each Job of each application
- $P_s = T_c - T_s - L_s$
- Equal to the negation of the remaining time
- Larger value has higher priority

2, Find the biggest P_s in each pool as the representative value

3, Resources are allocated in order of the pool with the largest representative value
- A resource allocation unit is TaskSet, which is a group of tasks and is a component of the Job





Proposal: Latency-aware task scheduler

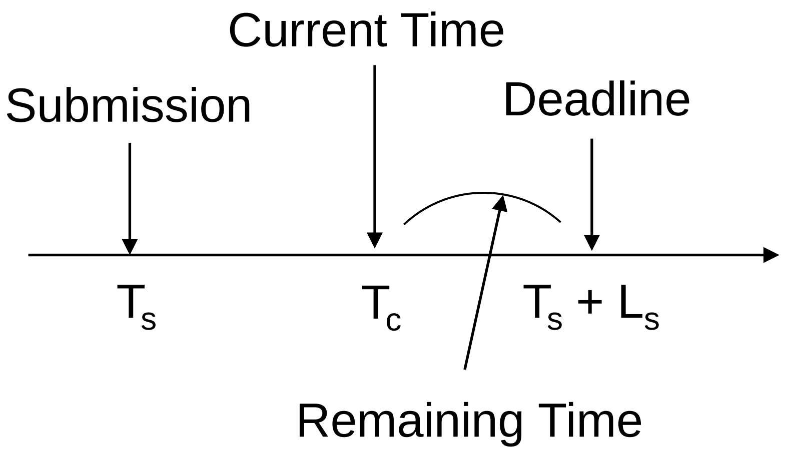
Priority based EDF (PT)

- Added potential priority p_s of the application s
  - e.g., anomaly detection may have higher priority than aggregation for visualization
  - The equation is modified as follows
  - $P_s = \frac{T_c - T_s}{L_s} p_s$

P_s	Priority of Job_s
T_c	Current Time
T_s	Submission time of Job_s
L_s	Latency SLA of Job_s

Processing time estimation (EST)

- Added estimated job execution time $F(I(s, T_s))$
  - $F()$ is an estimation function, and $I(s, T_s)$ is the input data rate
  - In this paper, $F()$ is defined through the measurement of the actual latency
  - The equation is modified as follows
  - $P_s = \frac{T_c - T_s + F(I(s, T_s))}{L_s} p_s$



Evaluation

■ Testbed Environment

- 5 servers Hadoop cluster with YARN (Specifications are shown in the table)
- Kafka as the data source and sink

CPU	Xeon E5-2620v4 (8Core) x 2
Memory	128 GB DDR4
Storage	15 TB of HDD, 128 GB of SSD
Network	10 Gb
OS	CentOS 6.9

■ Applications

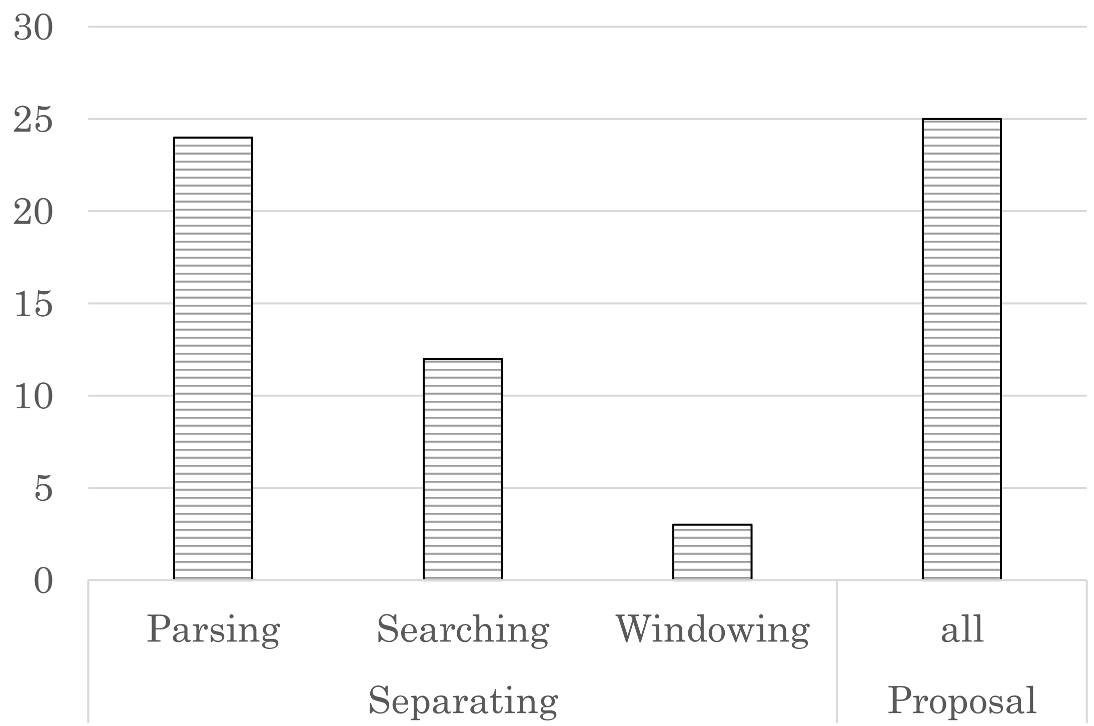
- 3 types of connected car applications (Parsing, Searching, and Windowing)
- 3 copies of each type application (total 9 applications)

■ Comparison with

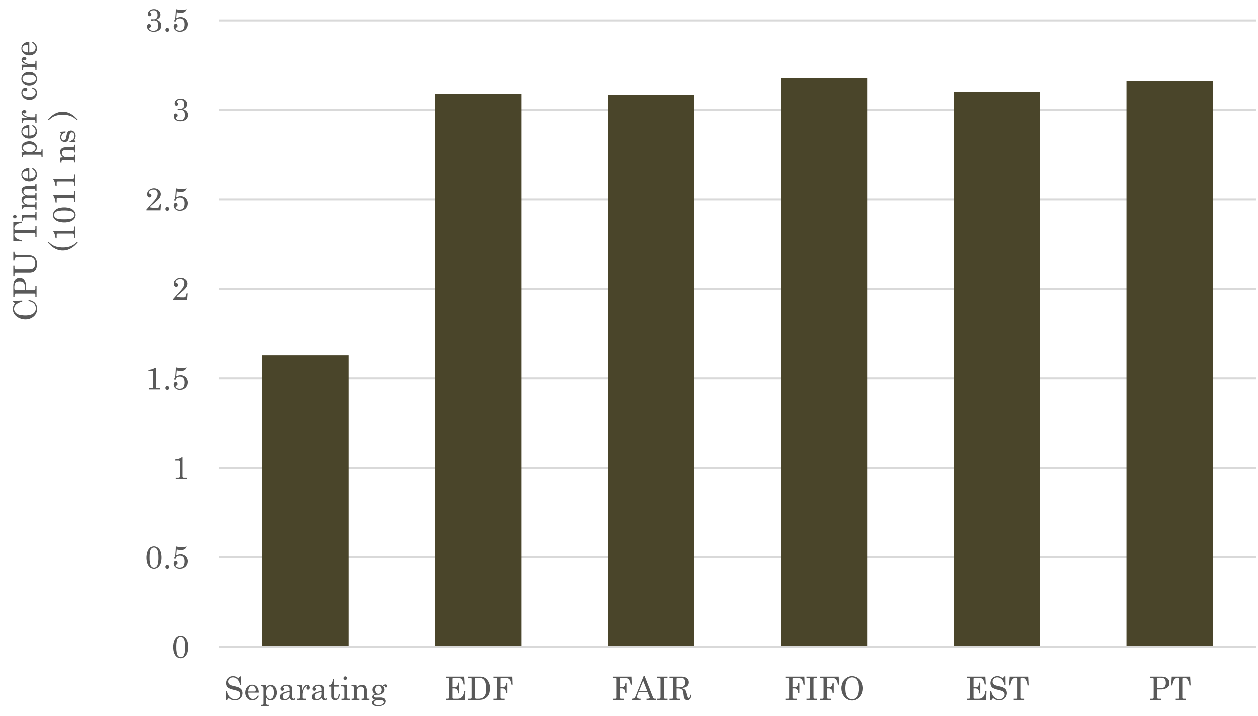
- Compared EDF
- Priority based EDF
- Process time estimation (EST)
- Default Spark (run applications separately)

Evaluation

Required minimum number of CPU cores to fulfil applications‘ SLAs



CPU time per core during experiment

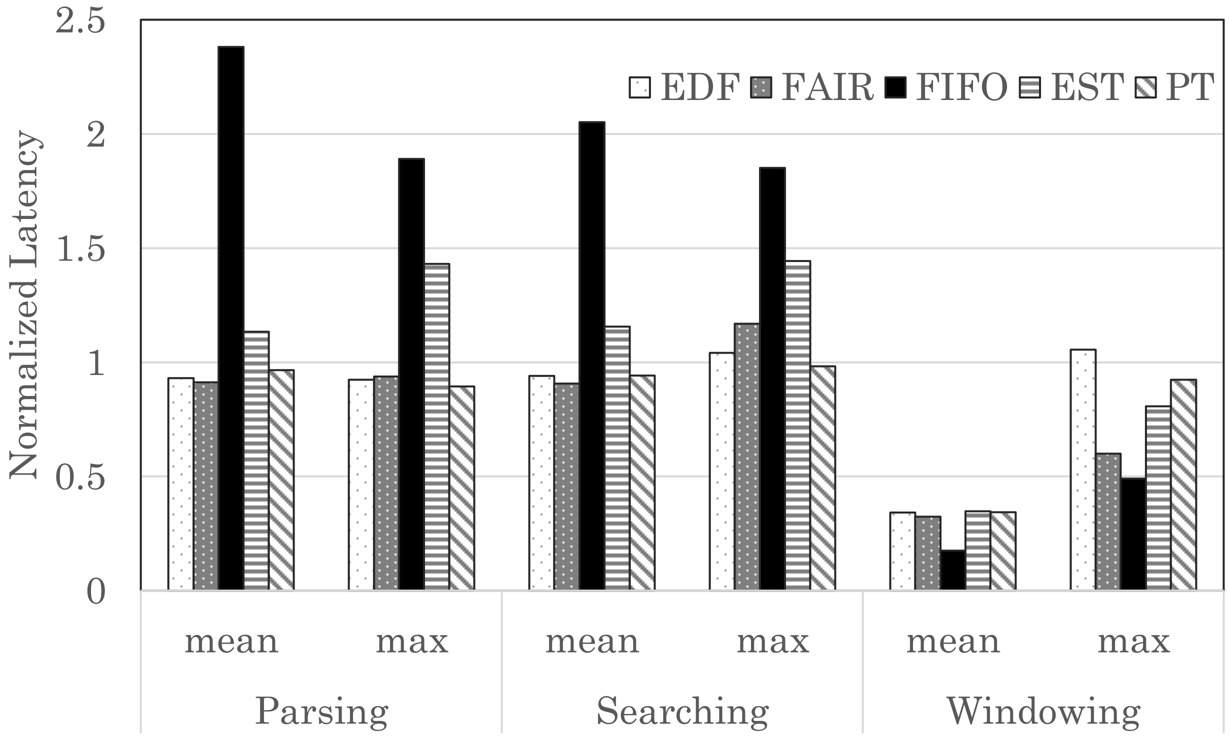


Same performance with 64% (25/39) CPU cores

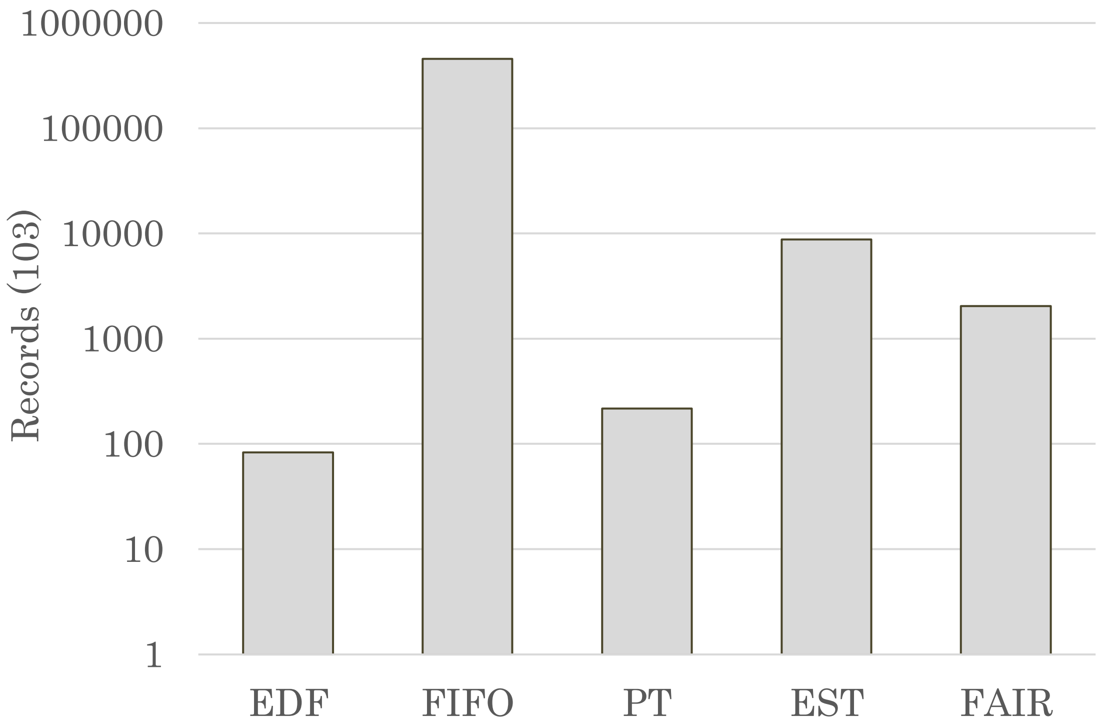
2x the utilization

Evaluation

The latency normalized by the latency of the default Spark



The number of records violated latency SLAs when the platform was overloaded



Achieved smaller latency

1/1000 SLA violation

Conclusion

■ Summary

- Goal: resource efficient stream processing
- Accommodating multiple applications
  - Scheduling in the same cluster enables quick reallocation and fine-grained control
- Latency-aware schedulers
  - Task-level granularity schedulers

■ Future Work

- Tradeoff between resource efficiency and isolation
- Consideration on other resources (e.g., memory)

KDDI Research

Thank you!

KDDI Research

Contact : yu-morisawa@kddi-research.jp
masaki-suzuki@kddi-research.jp
kitahara@kddi-research.jp