Online Resource Optimization for Elastic Stream Processing with Regret Guarantee

LIU Yang
Shanghai University
August 29, 2022

Contents

- Introduction of Distributed Stream Processing Systems
- Cloud Configuration Optimization and Related Works
- Formulate into a Two-level Online Optimization Framework
- Algorithm Design and Performance Guarantee
- Evaluation Based on Kubernetes-Implementation

Introduction of Distributed Stream-processing Systems

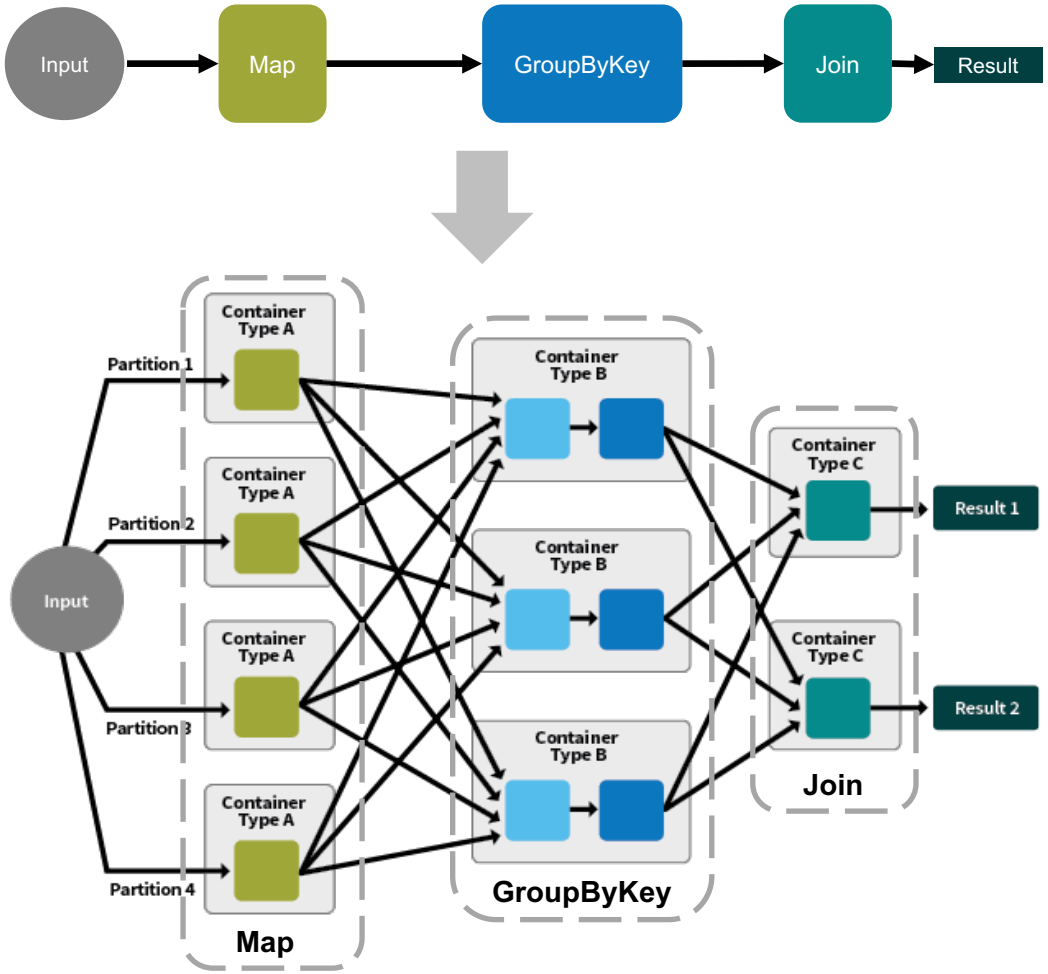


Figure 1. A data stream graph example.

- Stream processing application process incoming data via a data stream graph.
- Node is an operator which can run on multiple servers in parallel.
- Edge indicates how tuples are passed around operators/ servers.
- The data stream graph provides a logic view of the data transformation.

Cloud Config. Optimization for Elastic Stream-processing

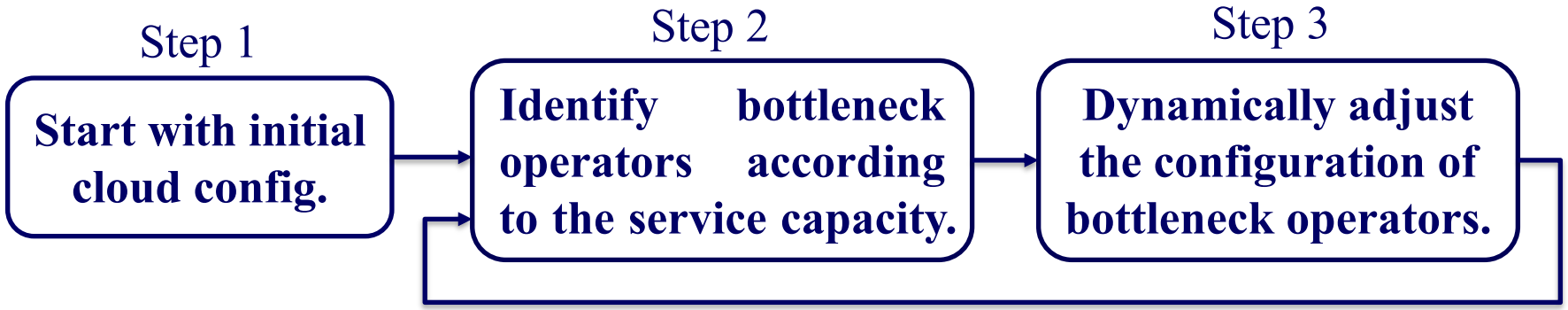


Figure 1. Dragster Workflow

- Dragster: dynamically adjust cloud configuration for each operator to maximize the application streaming throughput with varying workloads under a budget constraint.

- Challenge :
  - The performance of an application critically depends on its data stream graph.
  - The non-trivial relationship between the performance and its resource configuration.
  - The incoming source data often varies considerably (i.e., elastic stream processing).

Related Work of Cloud Configuration Optimization Problem

- Rule-based algorithm
  - *Dhalion*[VLDB 2017] in Heron linearly increases the no. of executors for bottleneck operators according to the backpressure.
- Learning-based algorithm
  - ML and RL algorithms have been applied to optimize the latency in [ALPOS 2019, 2021].
  - Black-box online optimization algorithms, e.g., Bayesian optimization and hill climbing algorithm, appeared in [MASCOTS 2016] [SoCC 2017][HPDC 2014][Sigmetrics 2003].
  - They ignore the DAG information of the data stream graph.
- Model-based algorithm
  - *Re-storm*[J.Inf.Sci. 2015] defines the critical path and adjusts its config. to optimize the latency.
  - *DS2*[OSDI 2018] in Flink linearly increases the no. of executors according to the service capacity.
  - They assume the operator performance following a simple or fixed mapping with candidate config..

Related Work of Cloud Configuration Optimization Problem

Reference	Search Algorithm	Optimization Mechanism	Consider DAG Info	Handling Time-varying Workload	Performance Guarantee
Dynamic Allocation [Spark]	Exponentially increase the no. of executors and remove the idle one. Implemented in Spark	Rule-based	**No**	**Yes**	**No**
Dhalion [VLDB 2017]	Linearly increase the no. of executors and remove the idle one. Implemented in Heron.	Rule-based	**No**	**Yes**	**No**
BO4CO [MASCOTS 2016]	Use Bayesian optimization framework to search the cloud config. with the optimal latency.	Learning-based	**No**	**No**	**No**
Sinan [ALPOS 2021]	Use CNN+LSTM to predict the latency under the candidate cloud configurations.	Learning-based	**No**	**No**	**No**
Re-storm [Tsinghua 2015]	Define the critical path and iteratively update the config. among the critical path to minimize the latency.	Model-based	**Yes**	**Yes**	**No**
DS2 [OSDI 2018]	Linear increase the no. of executor according to the service capacity. Implemented in Flink.	Model-based	**Yes**	**Yes**	**No**
Dragster	Combine online-optimization and GP-UCB techniques to identify the bottleneck operator and update its configuration.	Model-based + Learning-based	**Yes**	**Yes**	$O(\sqrt{TlogT})$ sub-linear regret

Related Works

[Spark] Spark Dynamic Allocation. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-dynamicallocation.html.
[VLDB 2017] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. Proceedings of the VLDB Endowment 10, 12 (2017), 1825–1836.
[MASCOTS 2016] Pooyan Jamshidi and Giuliano Casale. 2016. An uncertainty-aware approach to optimal configuration of stream processing systems. In 2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 39–48.
[SoCC 2017] Muhammad Bilal and Marco Canini. 2017. Towards automatic parameter tuning of stream processing systems. In Proceedings of the 2017 Symposium on Cloud Computing. 189–200.
[Sigmetrics 2003] Tao Ye and Shivkumar Kalyanaraman. 2003. A recursive random search algorithm for large-scale network parameter configuration. In Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems. 196–205.
[HPDC 2014] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. 2014. Mronline: Mapreduce online performance tuning. In Proceedings of the 23rd international symposium on High-performance parallel and distributed computing. 165–176.
[J.Inf.Sci. 2015] Dawei Sun, Guangyan Zhang, Songlin Yang,Weimin Zheng, Samee U Khan, and Keqin Li. 2015. Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments. Information Sciences 319 (2015), 92–112.
[OSDI 2018] Kalavri, Vasiliki, et al. "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows." 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). 2018.
[ALPOS 2019] Gan Y, Zhang Y, Hu K, et al. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices[C]
[ALPOS 2021] Zhang, Yanqi, et al. "Sinan: Data-Driven, QoS-Aware Cluster Management for Microservices."

Formulate into a Two-level Online Optimization Framework

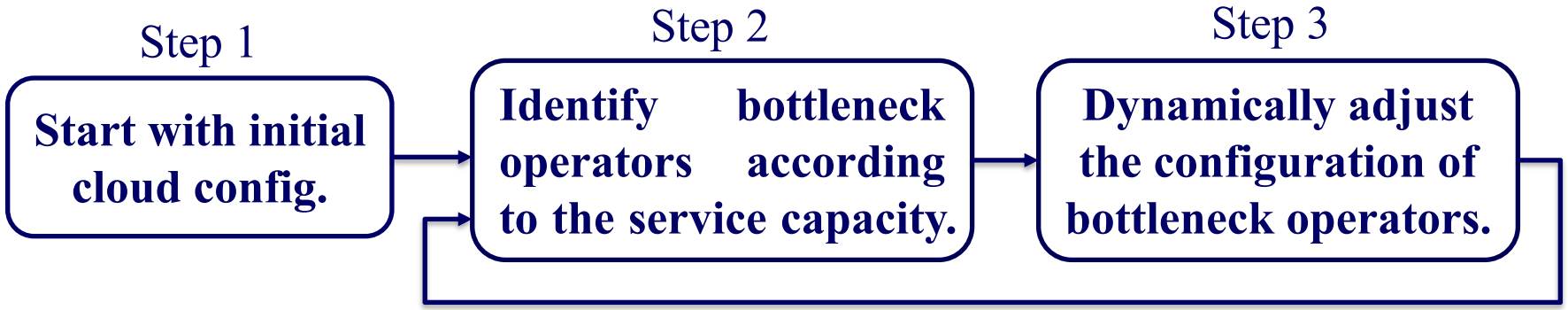
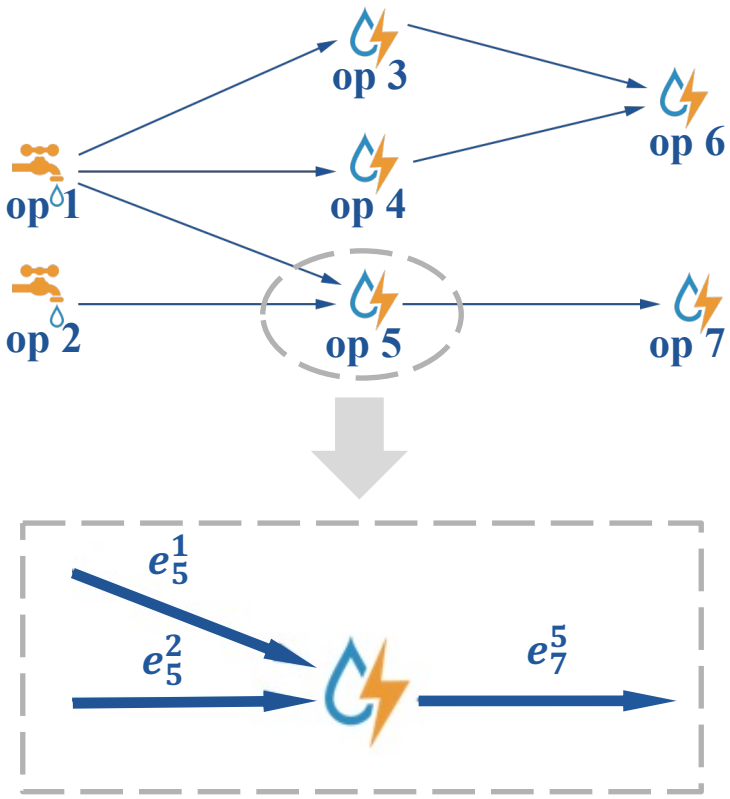


Figure 1. Dragster Workflow

- Identify the bottleneck operator according to the service capacity, i.e., processing rate
  - Downstream operator needs to consume all the received tuples
  - Model-based techniques: formulate into an online optimization problem
- Adjust the configuration of bottleneck operator
  - Allocate just enough computing resources to obtain the target service capacity, no wasting
  - Learning-based techniques: formulate into a multi-armed bandit problem

Level 1: Identify the Bottleneck Operator – Operator Wise

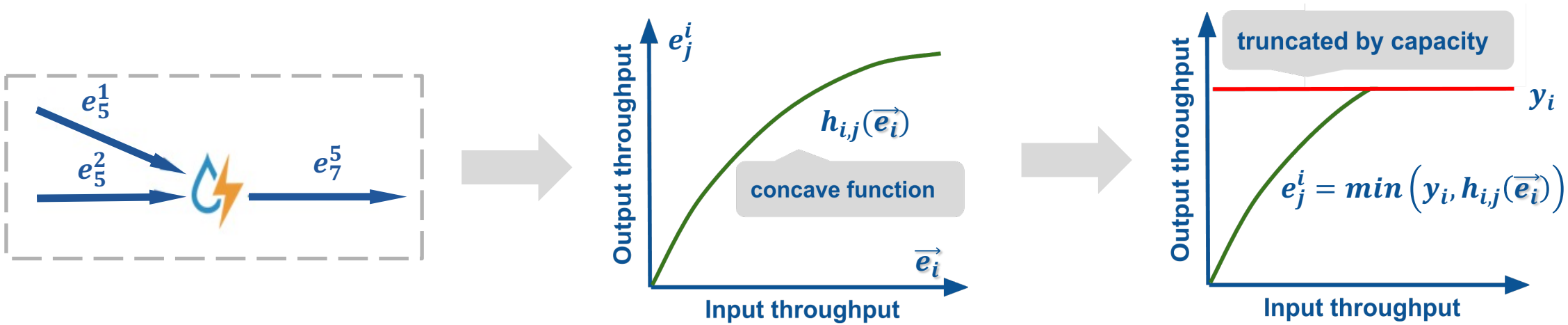


- Unlimited service capacity: operator i can process all the received tuples.

- Operator has multiple predecessors and successors.
- e_j^i: throughput from operator i to operator j.
- $\overrightarrow{e_i}$: the received throughput vector.

- The throughput function $h_{i,j}$ captures the input and output throughput mapping under unlimited service capacity case.

Level 1: Identify the Bottleneck Operator – Operator Wise

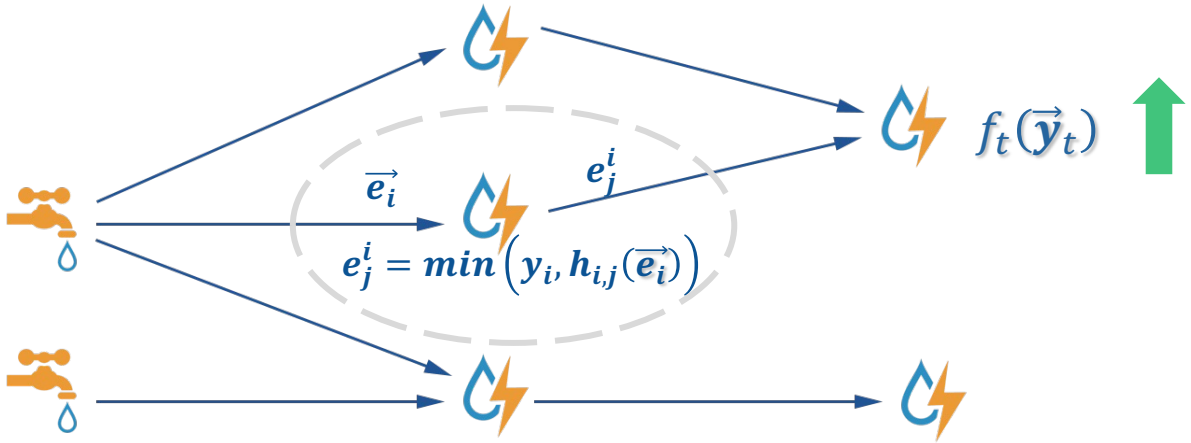


- Limited service capacity y_i: maximum processing rate of operator i.
- Service capacity y_i truncates the throughput function $h_{i,j}$:

$$e_j^i = \min\left(y_i, h_{i,j}(\vec{e_i})\right)$$

- $h_{i,j}$ can be either known beforehand or learned along with running the application (i.e., regression).

Level 1: Identify the Bottleneck Operator – Application Wise



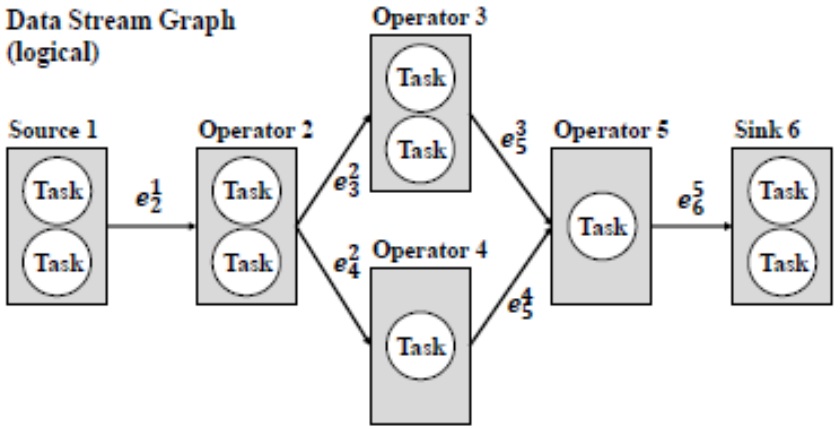
- The application throughput $f_t(\vec{y}_t)$ is a composition of the operator throughput function $h_{i,j}$, where $\vec{y}_t$ is the service capacity vector.
- Object: maximize the accumulated throughput. $\max_{\vec{y}_t} \sum_{t=1}^{T} f_t(\vec{y}_t)$
- Constraint: operator i can process all the received tuples over time T.

$$\sum_{t=1}^{T}\left(\sum_j h_{i,j}(\vec{e_i}) - y_i(t)\right) \leq 0$$

Bottleneck operator

- The solution is the target service capacity for each operator.

Level 1: Identify the Bottleneck Operator - Example




```
1  #Throughput function for application in Figure 2
2  num_of_source = 1
3  num_of_operator = 4
4  throughput_op2_1 = min(0.6 * capacity_op2, source1)
5  throughput_op2_2 = min(0.4 * capacity_op2, source1)
6  throughput_op3 = min(capacity_op3, 10000.0 * np.tanh(throughput_op2_1/10000.0))
7  throughput_op4 = min(capacity_op4, 10000.0 / ( 1.0 + np.exp(-throughput_op2_2/10000.0))
8  sink = min(capacity_op5, 1.5 * throughput_op3, 2.0 * throughput_op4)
```

Level 2: Adjust the Config. of Bottleneck Operator

- The service capacity y_i can not be set directly since it depends on its resource configuration $\boldsymbol{x}_i$ in an unknown manner.
- Take an online learning-based technique, Gaussian-Process multi-armed bandit.
- Assume the service capacity y_i following a Gaussian Process model.

$$y_i \sim GP(\mu_{i,t}(\boldsymbol{x}_i(t)), \sigma_{i,t}^2(\boldsymbol{x_i}(t)))$$

- Update the posterior distribution to capture the mapping between configuration and service capacity.

Two-Level Online Optimization Framework – Summary

Level 1: Identify the bottleneck operator

$$\max_{\{\boldsymbol{x}_i(t)\}} \quad \sum_{t=1}^{T} f_t(\boldsymbol{y}_t),$$

→ Goal: maximize the accumulated throughput.

$$s.t. \quad \sum_{t=1}^{T}\Big(\sum_{j\in S_i} h_{i,j}(\vec{\boldsymbol{e}_i}) - y_i(t)\Big) \le 0, \ \forall i > N,$$

→ Long-term constraint: operator buffer constraint.

- Physical meaning: the service capacity should be greater than the average incoming rate

---

Level 2: Adjust the configuration of the bottleneck operator

$$y_i \sim GP(\mu_i(\boldsymbol{x}_i), k_i(\boldsymbol{x}, \boldsymbol{x}_i)), \ \forall i > N,$$

→ The service capacity follows a GP model.

$$\sum_{i>N} \boldsymbol{x}_i(t) \le \boldsymbol{B}, \ \forall t.$$

→ Resource budget constraint.

Algorithm Design of Dragster

- Level 1: Use online optimization technique to determine the target capacity
  - Define the Lagrangian function: $L_t(\vec{\boldsymbol{y}}_t, \vec{\boldsymbol{\lambda}}_t) = f_t(\vec{\boldsymbol{y}}_t) + \sum_i \lambda_i \cdot l_i(y_i(t))$
  - Determine current time-slot capacity via online saddle point/ online gradient descent algorithm

$$\vec{\boldsymbol{y}}_t = argmax_{\vec{\boldsymbol{y}}} \; L_{t-1}(\vec{\boldsymbol{y}}, \vec{\boldsymbol{\lambda}}_{t-1}) \quad / \quad y_i(t) = y_i(t-1) + \eta \cdot dL_{t-1}(\vec{\boldsymbol{y}}_{t-1}, \vec{\boldsymbol{\lambda}}_{t-1})/dy_i$$

  - Update the dual variable: $\lambda_i(t) = max\{0, \lambda_i(t-1) + \gamma \cdot l_i(y_i(t))\}$
- Level 2: Extend the GP-UCB algorithm to adjust the configuration
  - The service capacity y_i following a Gaussian Process model: $y_i \sim GP\left(\mu_{i,t}(\boldsymbol{x}_i(t)), \sigma^2_{i,t}(\boldsymbol{x}_i(t))\right)$.
  - Adjust the configuration $\boldsymbol{x}_i(t)$ to track the target capacity $y_i(t)$.

$$\boldsymbol{x}_i(t) = argmax_{\boldsymbol{x}} - \left|\mu_{i,t-1}(\boldsymbol{x}) - y_i(t)\right| + \beta_{t-1} \cdot \sigma^2_{i,t-1}(\boldsymbol{x})$$

- Performance Guarantee
  - If the throughput function $h_{i,j}$ is known beforehand or can be learned accurately enough (i.e., $\left|h_{i,j}(\vec{e_i}) - \tilde{h}_{i,j}(\vec{e}_i)\right| = o(1/\sqrt{t})$) , sub-linearly increasing dynamic regret and dynamic fit (i.e., $O(\sqrt[2]{\log(\delta)\, T(logT)^{d+2}})$) can be satisfied with $\geq 1 - 1/\delta$ probability.

Performance Guarantee of Dragster

- **Theorem 1:** Let $\gamma = 1/\sqrt{t}$ and $\beta_t = 2\log(t^2\pi^2\delta/6)$. If the throughput function $h_{i,j}$ is given beforehand, sub-linearly increasing dynamic regret (i.e., $O(\sqrt[2]{\log(\delta)\, T(logT)^{d+2}})$) and dynamic fit (i.e., $O(\sqrt[2]{\log(\delta)\, T(logT)^{d+2}})$) can be satisfied with $\geq 1 - 1/\delta$ probability.

- **Theorem 2:** If the throughput function $h_{i,j}$ is learned along with running the application, the same order dynamic regret and dynamic fit can be achieved under:

$$\left|h_{i,j}(\vec{e_i}) - \tilde{h}_{i,j}(\vec{e}_i)\right| = o(1/\sqrt{t})$$

- **Proof Intuition**
  - Regret is due to the lack of knowledge of i) the service capacity function $y_i(\boldsymbol{x}_i)$ and ii) the application throughput function $f_t(\vec{\boldsymbol{y}}_t)$ beforehand.
  - Use the extended Gaussian-Process UCB algorithm to capture $y_i(\boldsymbol{x}_i)$ and lead to the term $O(\sqrt[2]{\log(\delta)\, T(logT)^{d+2}})$ in dynamic regret and fit.
  - Use the online saddle point/ online gradient point algorithm to capture $f_t(\vec{\boldsymbol{y}}_t)$ and lead to the term $O(\sqrt[2]{T})$ in dynamic regret and fit.

Implement Dragster for Apache Flink over Kubernetes

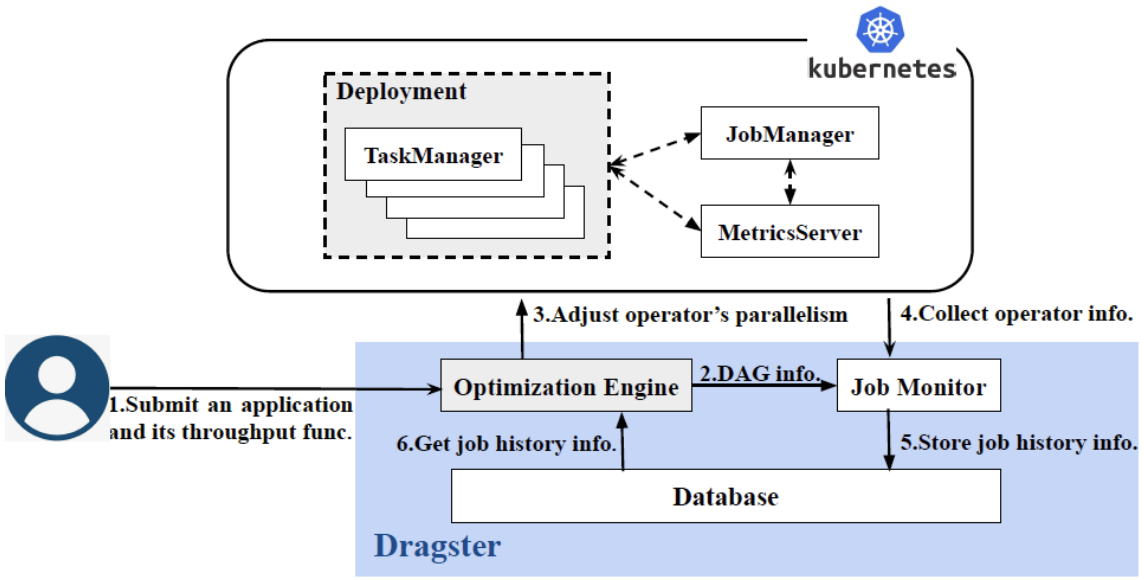


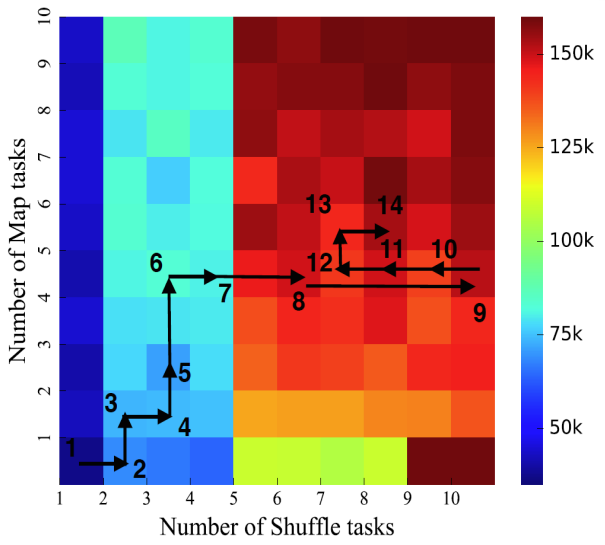
Figure 1. System architecture of Dragster for Apache Flink applications over Kubernetes

- Use Dragster to optimize:
  - Every 10 mins, adjust the no. of executors for each operator.
  - Re-configuring via Checkpoint takes among 30s.

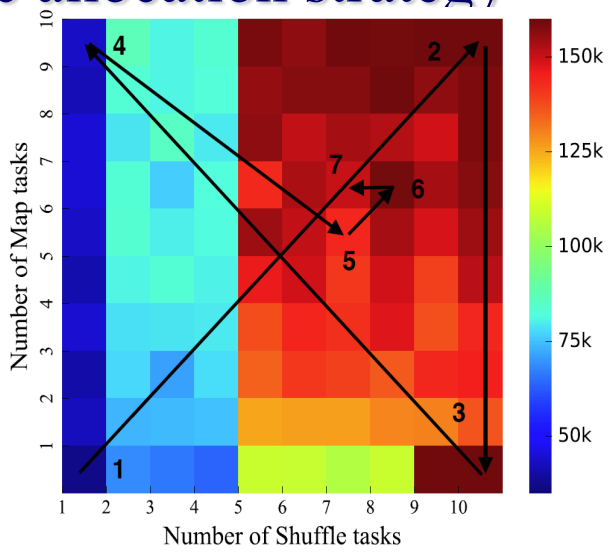
- Three components:
  - **Database** stores history information, including resource configuration, throughput and etc.
  - **Optimization Engine** implements Dragster algorithm to adjust the current time-slot config..
  - **Job Monitor** uses REST API and Kubernetes metrics server to collect running time information.

Insight of Dragster

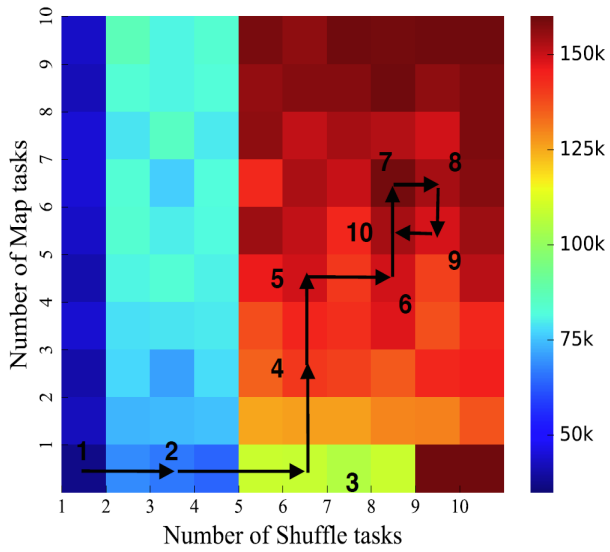
- Three schemes
  - The rule-based algorithm, Dhalion: linearly increase the no. of executors.
  - Dragster with online saddle point: set the service capacity as the last time-slot optimum.
  - Dragster with online gradient descent: gradually adjust the service capacity.
- Sequence of chosen configurations running Streaming WordCount
- Provide customized dynamic allocation strategy



(a) Throughput under the rule-based algorithm. 14 runs



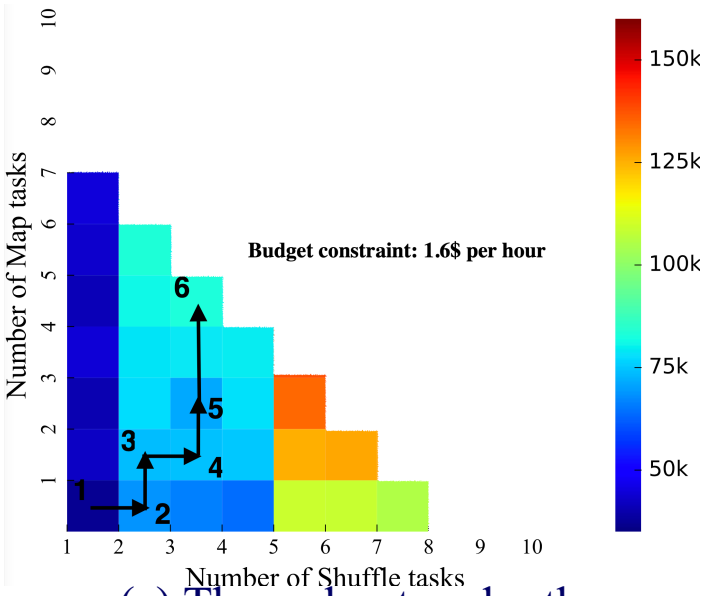
(b) Throughput under Dragster with online saddle point. 7 runs



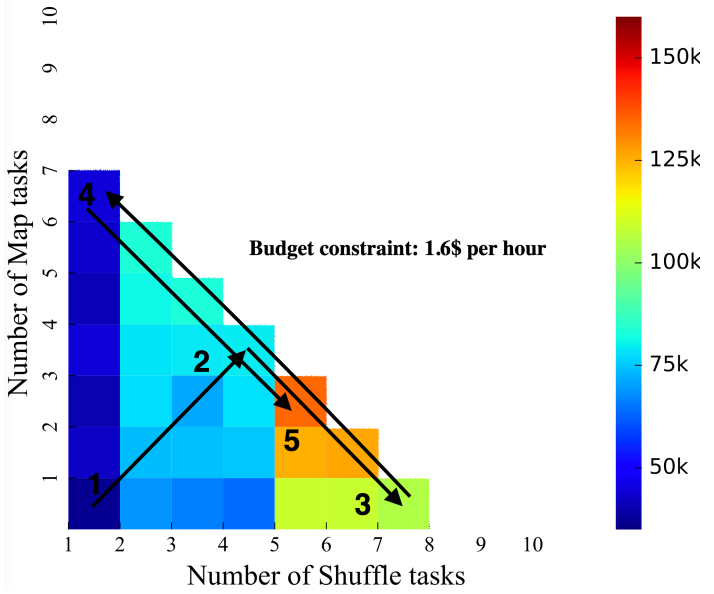
(c) Throughput under Dragster with online gradient descent. 10 runs

Insight of Dragster

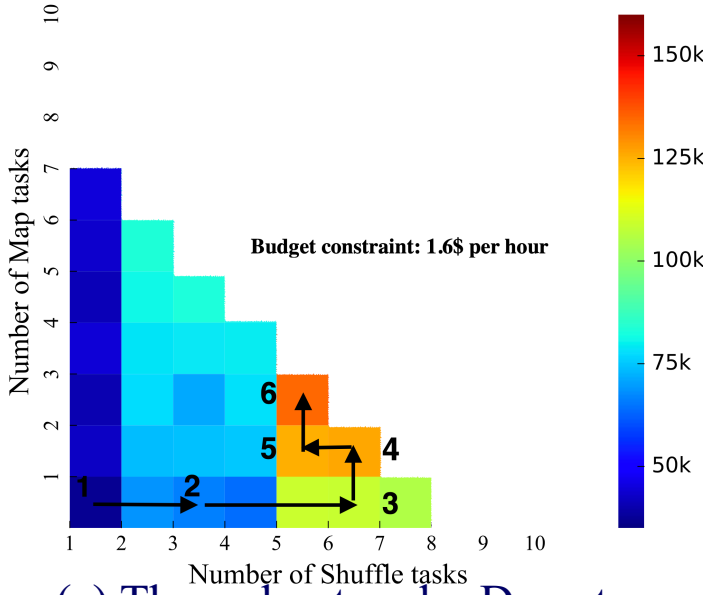
- Sequence of chosen configurations running WordCount under a budget constraint.
- Dragster performs even more better, because it uses the DAG information
- Dhalion allocates most of the resources to the mapper operator and cannot balance the resources for the shuffle operator.



(a) Throughput under the rule-based algorithm.



(b) Throughput under Dragster with online saddle point.



(c) Throughput under Dragster with online gradient descent.

Performance Evaluation under Workload Changing

- Streaming WordCount contains 2 operators.
- Quickly converge under workload changing, especially the fluctuation happens before.
- Dragster with saddle point converges 2.0X-speedup and processes 16.3% more tuples.
- Dragster with gradient descent converges 1.8X-speedup and processes 17.4% more tuples.

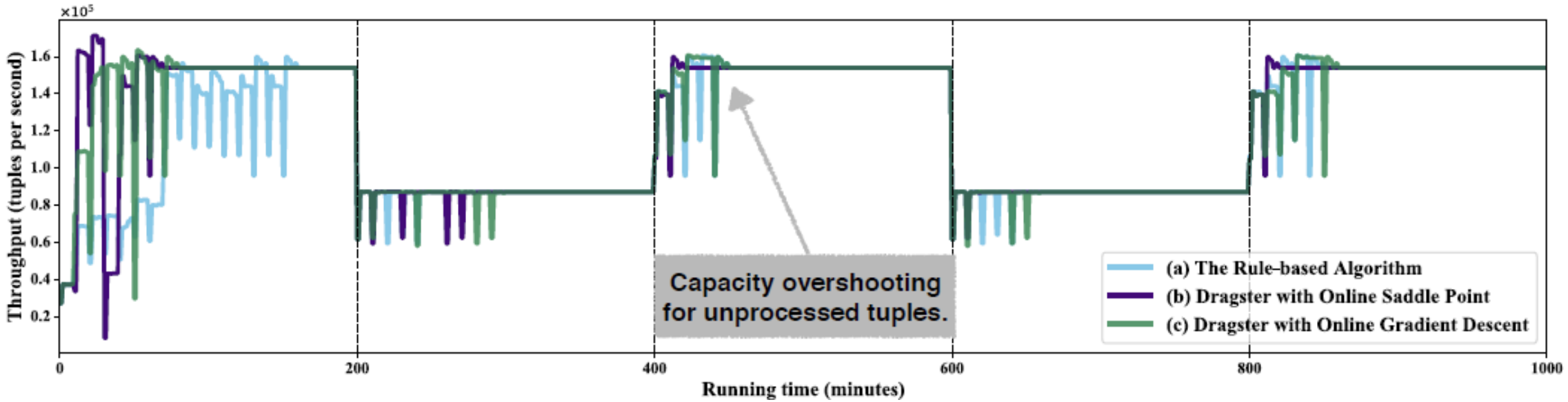


Figure 1. Throughput achieved by Dragster under workload changing running streaming WordCount.

Performance Evaluation under a Diverse Set of Workloads

- 6 application: Group, AsynclIO, Join, Window, WordCount, Yahoo streaming benchmark.
- The number of operator ranges from 1 to 6. Two different source inputting rate.
- Dragster performs more and more better with the increasing number of operators.

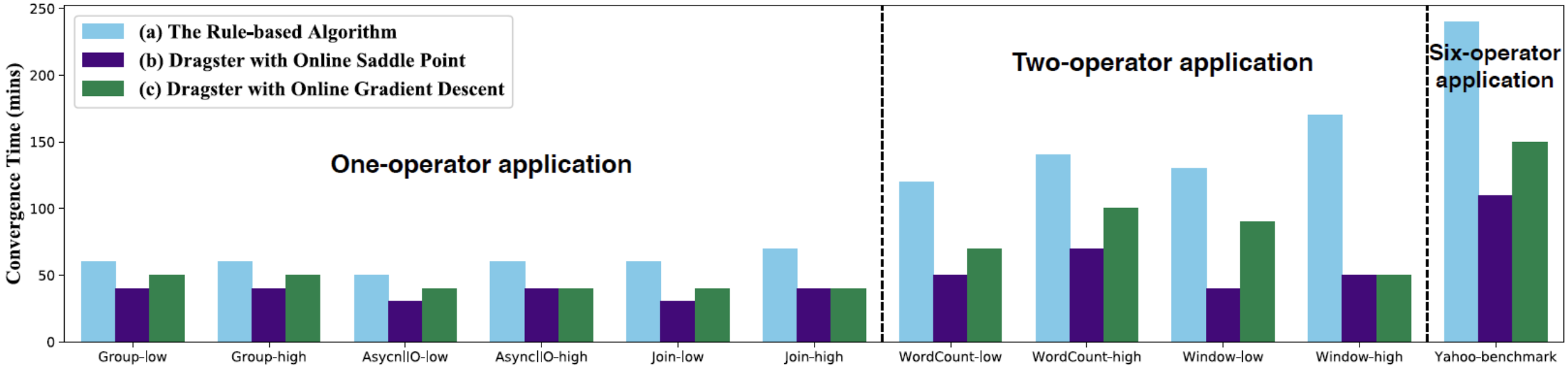


Figure 1. Convergence time under a diverse set of workloads.

Performance Evaluation – More Complex Application

- Yahoo streaming benchmark: 6 operators, 1 million candidate configurations.
- Dragster with saddle point converges 2.2X-speedup and processes 23.7% more tuples.
- Dragster with gradient descent converges 1.6X-speedup and processes 25.8% more tuples.

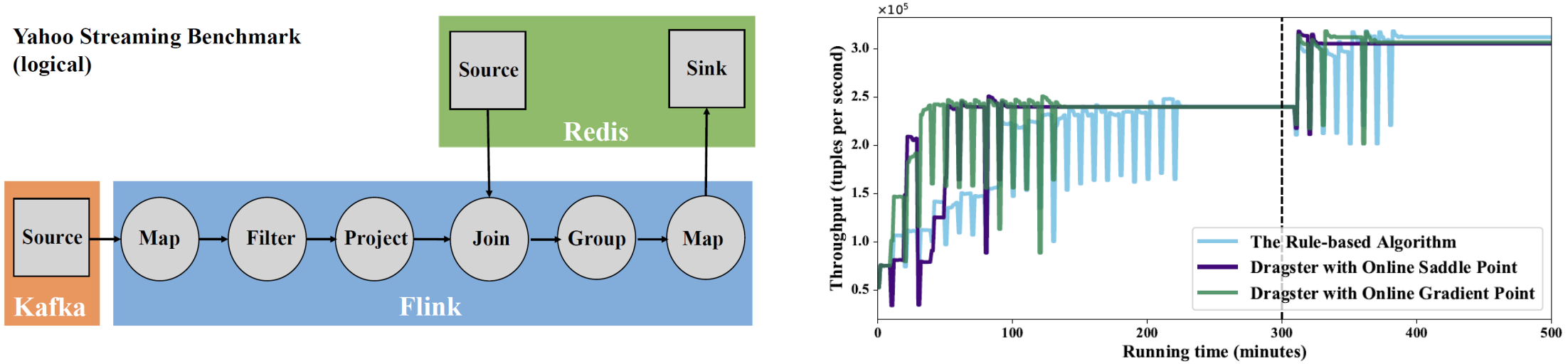


Figure 1. Throughput achieved by Dragster under workload changing running Yahoo streaming benchmark.

Thank You