基于Kubernetes的大数据系统、微服务系统资源配置动态优化

姓名：　刘洋
部门：　上海大学　-　计算机学院
日期：　2022年12月12日

个人简介

教育及工作经历：

- 上海大学 计算机学院 讲师 2021.11 - 至今
- 字节跳动-基础架构-字节云 后端开发工程师 2021.05 - 2021.11
- 香港中文大学 博士
  - 工程学院 信息工程系 导师：Wing Cheong Lau 2016.08 - 2021.05
- 上海交通大学 本科
  - 致远学院 数理科学班 2010.09 - 2013.06
  - 电子信息与电气工程学院 计算机科学与工程系 2013.09 - 2016.06

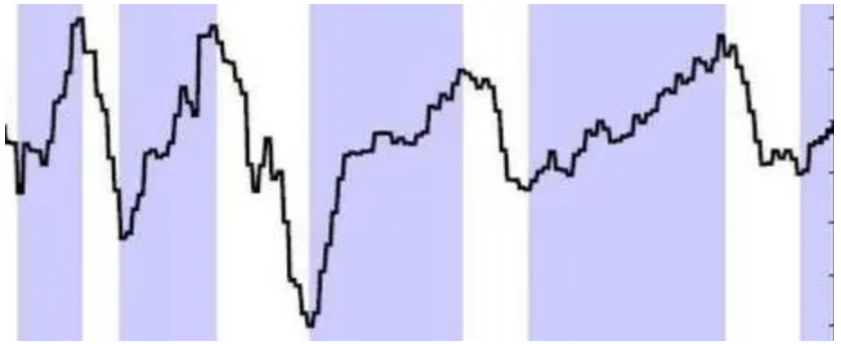
研究方向：

- 云计算系统，AI for Systems，发表：INFOCOM、ToN、ICDCS、ICPP、IJCAI

主持项目：

- 国自然，青年科学基金，“基于在线学习的流处理系统资源配置动态优化” 2023.01 - 2025.12
- 上海市浦江人才计划 - A类 科技研发 2022.10 - 2024.09
- 上海市领军人才（海外）

云计算中心现状



实时型作业和计算型作业，混合部署：

- 实时作业：微服务时延要求 -> 请求过量资源 -> 资源使用率低 -> 当作低质量资源超售 资源劣化
- 计算作业：Spark、 ML 时延无要求 + 需重复运行 -> 使用低质量、廉价资源

资源使用率“波峰波谷”明显：

- 20% – 70%的资源使用率， CPU使用率超过60% -> 时延急剧提高

智能调度 解决方案：实时作业 弹性扩缩容；计算作业 削峰填谷

研究方向 - 云计算系统智能调度

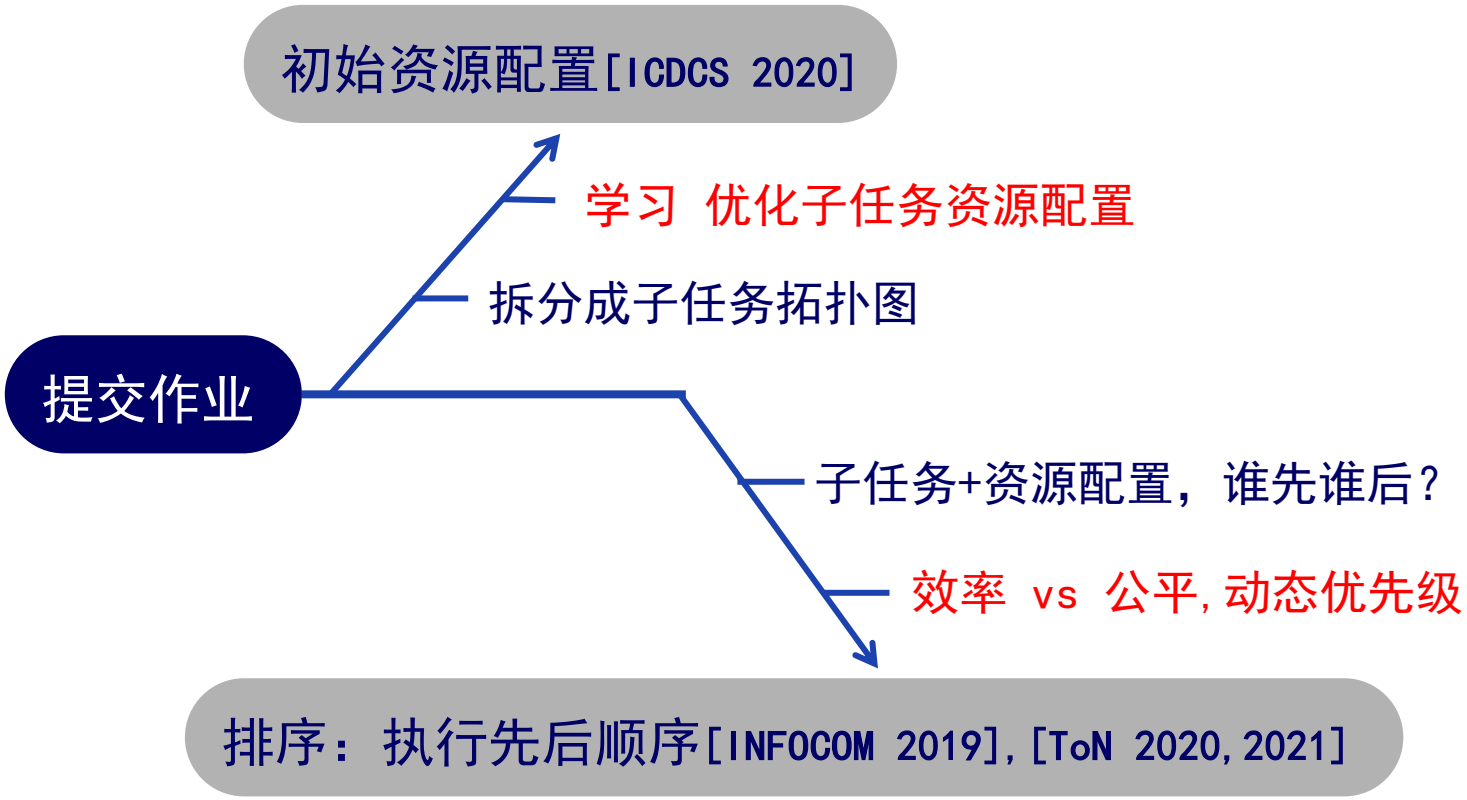
通过调节容器资源配置、执行顺序、部署位置，为大数据中心提高吞吐量、降低时延、提高资源利用率、减少能耗

研究方向 - 云计算系统智能调度

通过调节容器资源配置、执行顺序、部署位置，为大数据中心提高吞吐量、降低时延、提高资源利用率、减少能耗

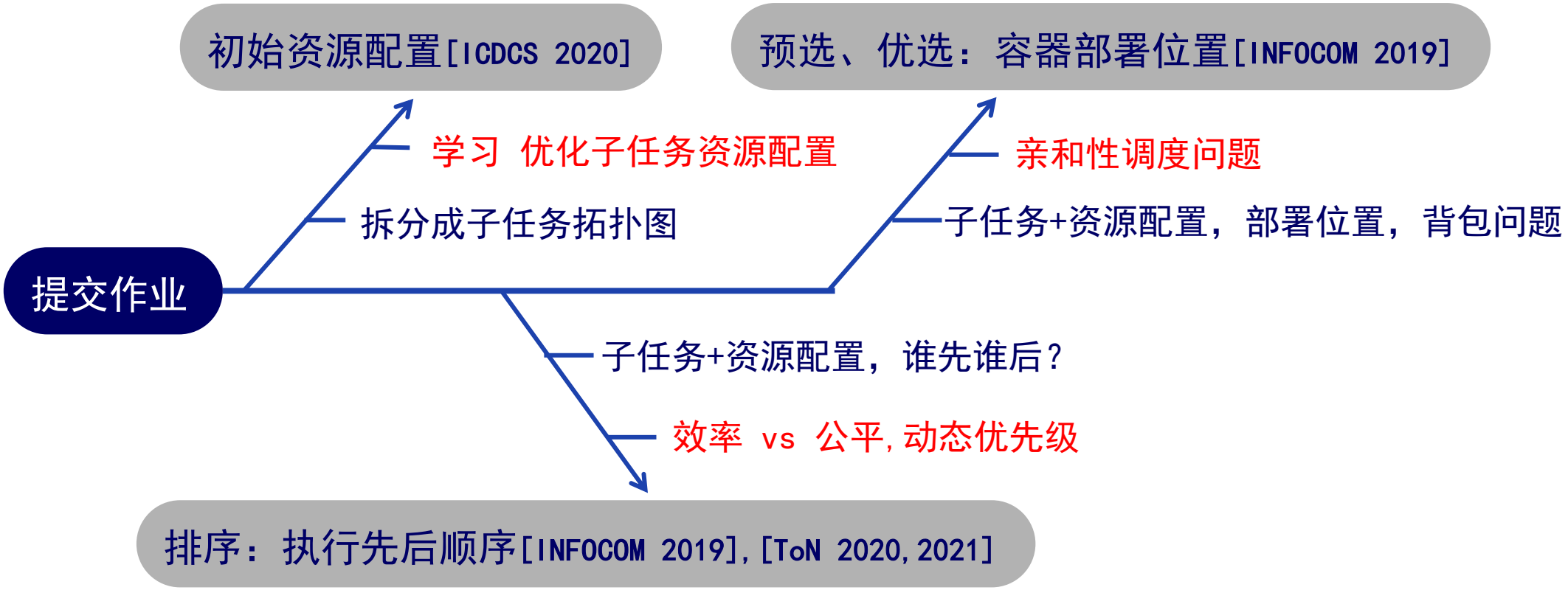
研究方向 - 云计算系统智能调度

通过调节容器资源配置、执行顺序、部署位置，为大数据中心提高吞吐量、降低时延、提高资源利用率、减少能耗



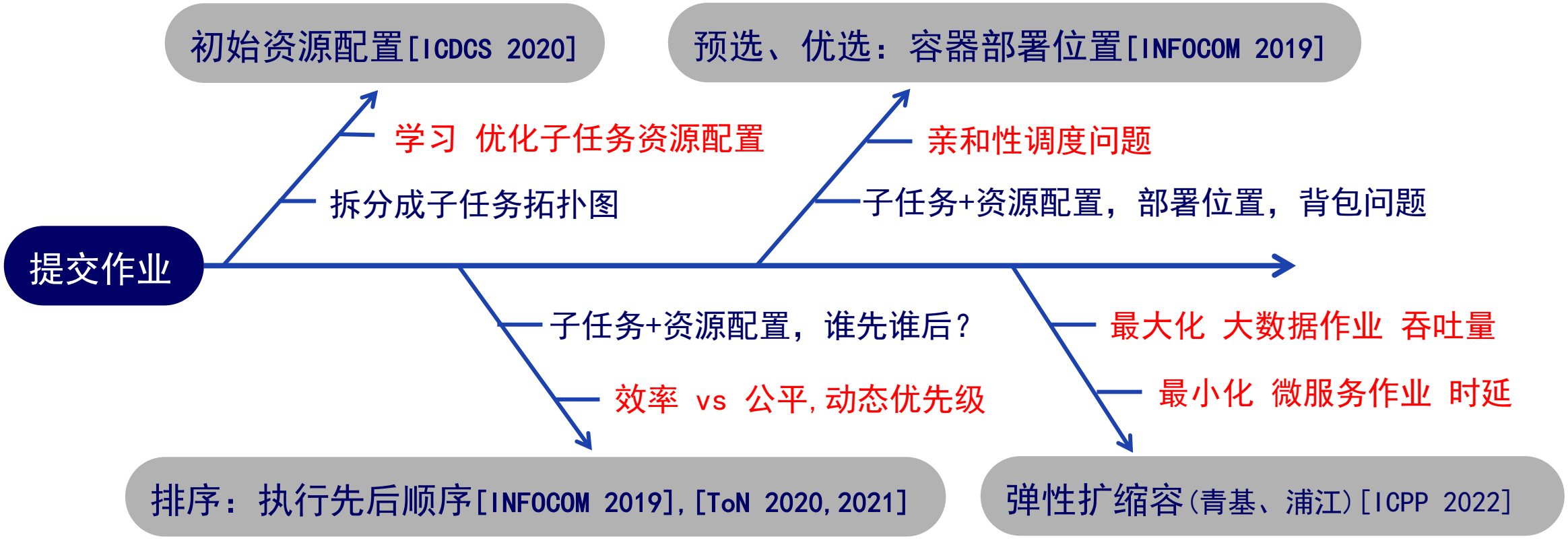
研究方向 - 云计算系统智能调度

通过调节容器资源配置、执行顺序、部署位置，为大数据中心提高吞吐量、降低时延、提高资源利用率、减少能耗



研究方向 - 云计算系统智能调度

通过调节容器资源配置、执行顺序、部署位置，为大数据中心提高吞吐量、降低时延、提高资源利用率、减少能耗



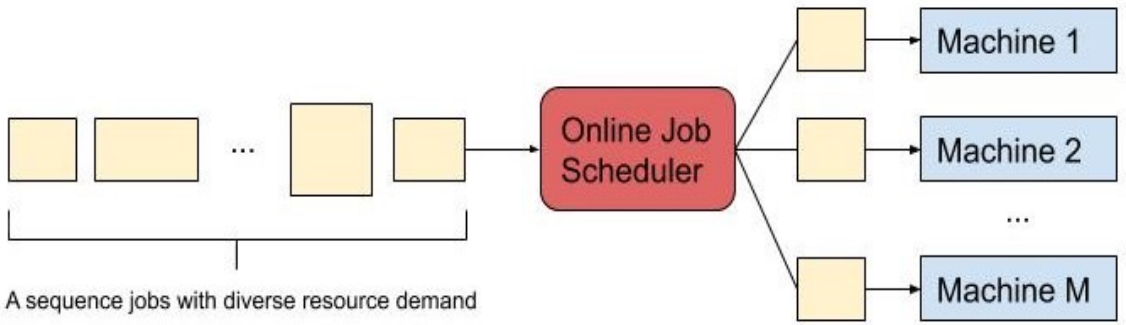
目录

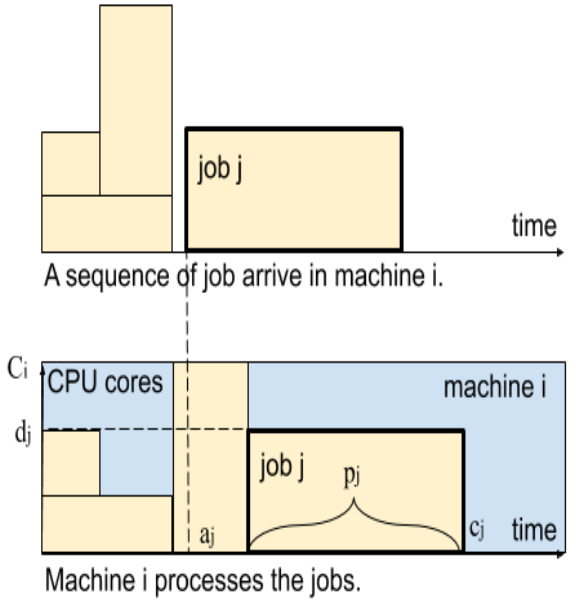
- OCORP，INFOCOM 2019、ToN 2021，批处理作业 Hadoop Yarn 在线调度
- Accordia，ICDCS 2020、TPDS 2022， 批处理作业 Spark 资源配置动态优化
- Dragster，ICPP 2022， 流处理作业 Flink 最大化吞吐量，自动弹性扩缩容
- DRAM，WWW 2023，微服务作业 最小化时延，自动弹性扩缩容

数学的模型 -> 直觉的结果 + 可分析的性能

OCORP: Online Job Scheduling with Resource Packing on a Cluster of Heterogeneous Servers

Introduction of Online Job Scheduling Problem





- C_i: machine i CPU resource ;
- a_j: job j arriving time;
- d_j: job j CPU resource demand;
- p_j: job j processing time;
- c_j: job j; completion time

- A sequence of jobs keep arriving in a cluster. Scheduler allocates jobs to machines.
- Machine i can provide C_i CPU cores (extended system model considers the multi-dimension resource case).
- Job j arrives at time slot a_j . It needs p_j units of time to finish, if the demand of d_j CPU cores is fully satisfied. c_j is the completion time of job j.

Basic System Model of OCORP

- Goal: minimize the overall $L_k - norm$ job flowtime: $\min \sum_{j=1}^{N} (c_j - a_j)^k$
- Varaible: CPU allocation rate $x_j^i(t) \in \{0,1\}$, whether job j works in machine i at time slot t.
- CPU resource constraint: the total demand of CPU cores cannot exceed the capacity of machine i.

$$\sum_{j=1}^{N} x_j^i(t) d_j \leq C_i, \quad \forall i, t$$

- Job completion constraint (long-term constraint): guarantee the completion of all the jobs before time slot T.

$$\sum_{t=1}^{T} \sum_{i=1}^{M} x_j^i(t) \geq p_j \quad \& \quad c_j \leq T, \quad \forall i, t$$

批处理任务的在线调度问题 - 基础模型

$$\min_{\{x_j^i(t)\}} \sum_{j=1}^{N}(c_j - a_j)^k$$

目标函数：最小化总体完成时间的 $L_k - norm$。

$$s.t. \quad x_j^i(t) \in \{0,1\}, \quad \forall i,j,t$$

$$\sum_{i=1}^{M} x_j^i(t) \le 1, \quad \forall j,t$$

自变量：在时刻 t，任务 j 是否运行在机器 i 中。

$$\sum_{j=1}^{N} x_j^i(t) d_j \le C_i, \quad \forall j,t$$

机器能够提供，所有任务运行所需 CPU。

$$\sum_{t=1}^{T}\sum_{i=1}^{M} x_j^i(t) \ge p_j, \quad \forall j,t$$

$$c_j \le T, \quad \forall j,t$$

保证任务 j 在时间 c_j 完成。

- 比多纬背包问题更复杂，NP-hard，无法求解。

批处理任务的在线调度算法 – 经典算法

- 最短剩余作业优先(SRPT) [Infocom 2013 - 2017]:
  - 不可并行的情况下，单个机器的最优算法。
  - 把任务发送到等待队列最短的机器上，单个机器上SRPT。

→ 饥荒 starving， 一维资源

- 主导资源公平(DRF)[NSDI 2011]:
  - 给用户公平的分配资源，使得主导资源所占份额相同。
  - 满足四个公平特性：sharing-incentive， strategy-proof， envy-free， Pareto-efficient。

→ 效率差，总体完成时间长

- Tetris, the-state-of-art heuristic scheduler [Sigcomm 2014, Berkeley]:
  - Sorting: 先按照DRF给所有用户的任务排序，再对前 $\alpha\%$ 的任务SRPT。
  - Packing: 按照 $\overrightarrow{\text{任务所需资源}} \cdot \overrightarrow{\text{机器剩余资源}}$ 分发任务到机器。

Related Works of Job Scheduling Problem

[Infocom 2017] Tan, Haisheng, et al. "Online job dispatching and scheduling in edge-clouds." in IEEE Infocom, 2017.
[Perform. Evaluation 2017] Gebrehiwot, Misikir Eyob, Samuli Aalto, and Pasi Lassila. "Energy-aware SRPT server with batch arrivals: Analysis and optimization." in Performance Evaluation, 2017.
[Infocom 2014] Y. Yuan, D. Wang, and J. Liu, “Joint scheduling of MapReduce jobs with servers: Performance bounds and experiments,” in IEEE Infocom, 2014.
[IFIP Perform. 2013] M. Lin, L. Zhang, A. Wierman, and J. Tan, “Joint optimization of overlapping phases in MapReduce,” in IFIP Performance, 2013.
[Infocom 2013] Y. Zheng, N. Shroff, and P. Sinha, “A new analytical technique for designing provably efficient MapReduce schedulers,” in Proc. of IEEE Infocom, 2013.
[NSDI 2011] Ghodsi, Ali, et al. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types." in NSDI, 2011.
[TEAC 2015] Parkes D C, Procaccia A D, Shah N. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities[J]. ACM Transactions on Economics and Computation (TEAC), 2015.
[STOC 2014] S. Im, J. Kulkarni, and K. Munagala, “Competitive algorithms from competitive equilibria: non-clairvoyant scheduling under polyhedral constraints,” in STOC, 2014.
[APPROX-RANDOM 2016] S. Im, J. Kulkarni, B. Moseley, and K. Munagala, “A competitive flow time algorithm for heterogeneous clusters under polytope constraints,” in APPROX-RANDOM, 2016.

基于在线优化的集群调度算法 - 转化为在线凸优化问题

$$\min_{\{x_j^i(t)\}} \sum_{j=1}^{N}(c_j - a_j)^k$$

$$s.t. \quad x_j^i(t) \in \{0,1\}, \quad \forall i,j,t$$

$$\sum_{i=1}^{M} x_j^i(t) \le 1, \quad \forall j,t$$

$$\sum_{j=1}^{N} x_j^i(t) d_j \le C_i, \quad \forall j,t$$

$$\sum_{t=1}^{T}\sum_{i=1}^{M} x_j^i(t) \ge p_j, \quad \forall j,t$$

$$c_j \le T, \quad \forall j,t$$

使用分时任务完成时间进行近似，把完成时间拆分到每个时刻，并保证他们对时间的累积求和 和原本相似。

对CPU分配比例 $x_j^i(t)$ 进行连续化放松。

把长时约束拆分到每个时刻，想要任务在时间 T 前完成，必须任务执行的平均速度达到要求。

$$\min_{\{x_j^i(t)\}} \sum_{t=1}^{T}\sum_{j:a_j>t}\sum_{i=1}^{M}\left(\frac{(t-a_j)^k}{p_j} - p_j^{k-1}\right)x_j^i(t)$$

$$s.t. \quad 0 \le x_j^i(t) \le 1, \quad \forall i,j,t$$

$$\sum_{i=1}^{M} x_j^i(t) \le 1, \quad \forall j,t$$

$$\sum_{j=1}^{N} x_j^i(t) d_j \le C_i, \quad \forall i,t$$

$$\sum_{i=1}^{M} x_j^i(t) \ge \frac{p_j}{T-a_j}, \quad \forall j,t$$

批处理任务的在线调度算法

- **分时任务完成时间:**

$$\min_{\{x_j^i(t)\}} \sum_{j=1}^{N}(c_j - a_j)^k$$

把完成时间拆分到每个时刻，并保证对时间的累积求和 和原本相似。

$$\sum_{t=1}^{T} f_t(x_j^i(t)) = \sum_{t=1}^{T}\sum_{j:a_j \geq t}\sum_{i=1}^{M}\left(\frac{(t-a_j)^k}{p_j} - p_j^{k-1}\right)x_j^i(t)$$

- 任务完成时 汇报完成时间。
- 任务到达、完成时，进行调度。
- 非凸优化，300节点，1h计算时间。

- 每一时刻，机器汇报任务完成比例。
- 周期化的在线调度。
- 在线凸优化，毫秒级计算时间。

- **一维资源下的解析解:**

$$\omega_j(t) = \left(\frac{(t-a_j)^k}{p_j} + p_j^{k-1}\right)/d_j \quad - \quad \lambda_j(t)/d_j$$

- 其中 $\omega_j(t)$ 给任务 j 提供动态优先级。
- 左半部分类似于SRPT，提供效率。 右半部分 $\lambda_j(t)$ 是等待时间，提供公平。

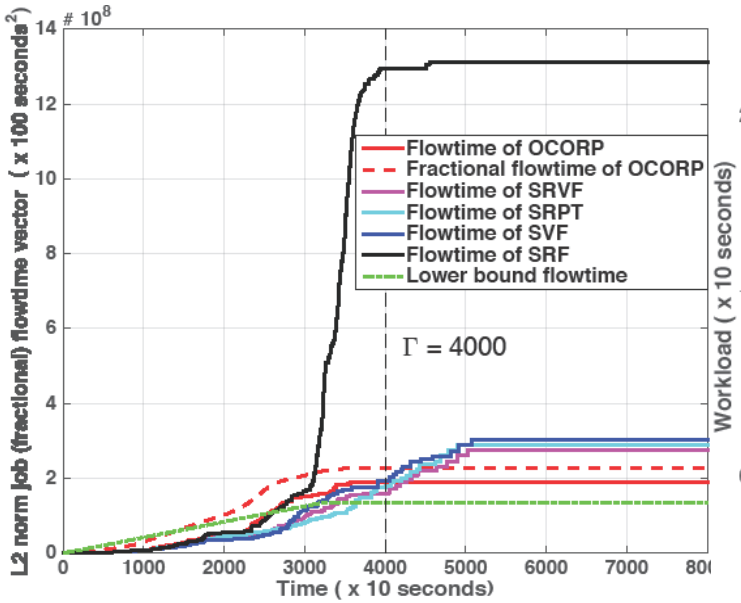
- **悔值函数 dynamic regret:**

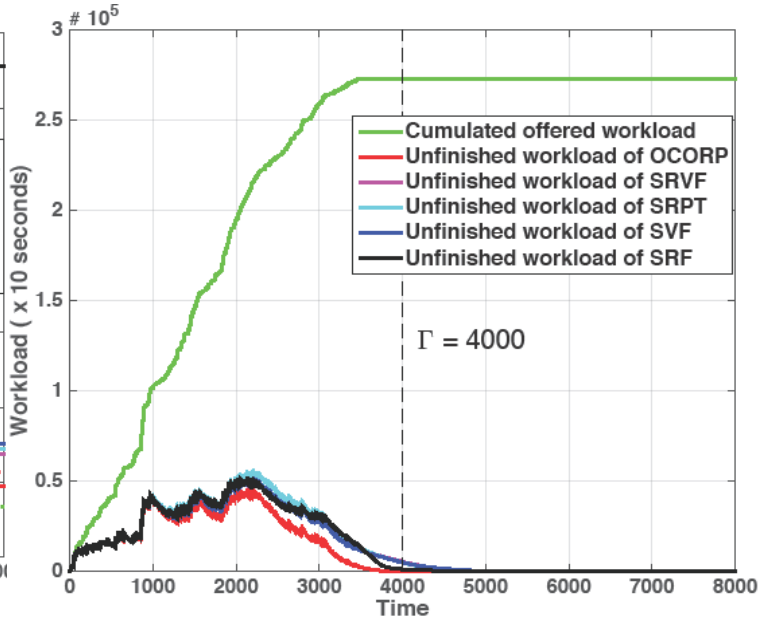
$$Reg_T = \sum_{t=1}^{T} f_t\left(x_j^i(t)\right) - OPT \leq OPT \times o\left(T^{\beta}\right), \qquad 0 \leq \beta \leq 1$$

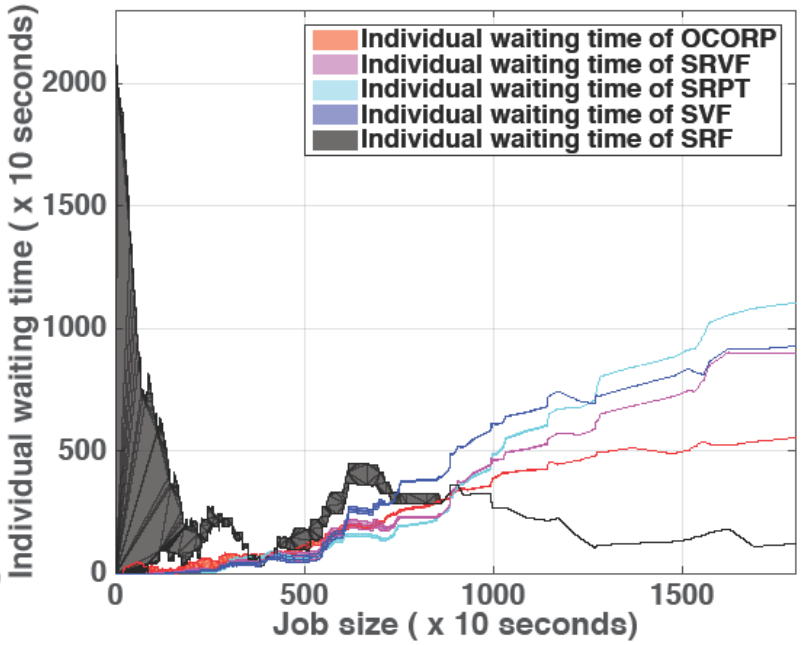
- **易实现，计算量小，最优理论结果。**

Performance Evaluation in Hadoop Yarn

- Evaluation Result of Basic System Model:
  - Driven by Google cluster-usage trace (6000 jobs arriving in 10 hours).
  - The curve of fractional job flowtime is close to that of job flowtime.
  - Reduce the overall job flowtime by 34% and the makespan by 20%.
  - Better fairness between small jobs and huge jobs.

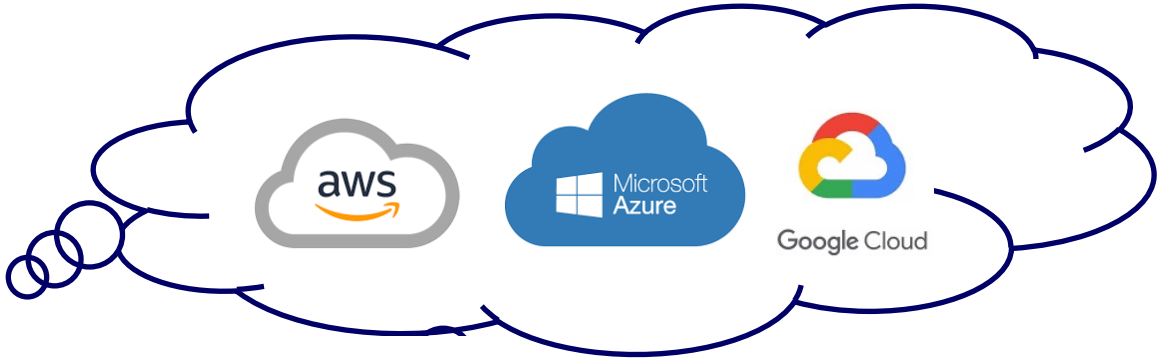






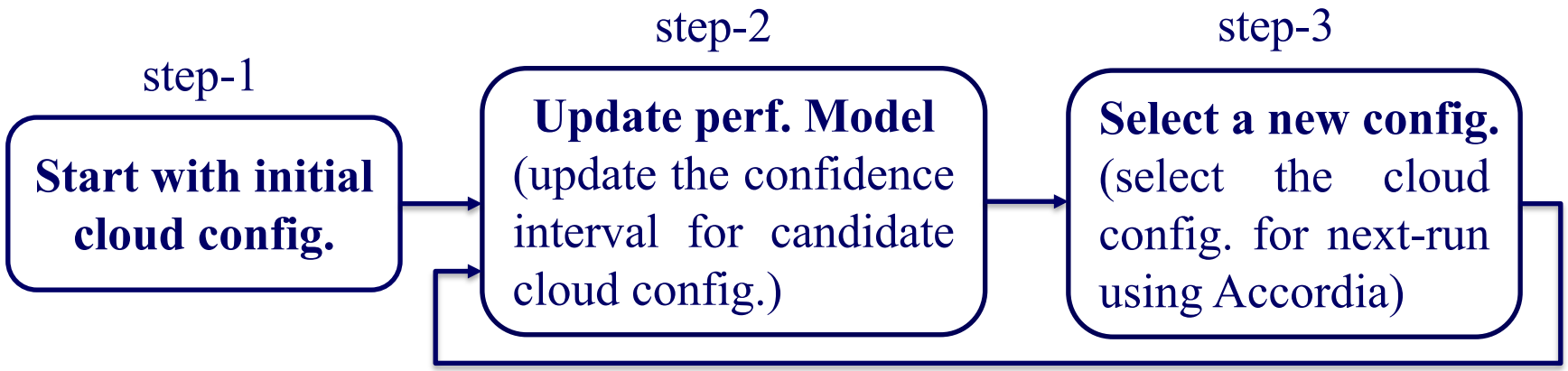
Accordia: Adaptive Cloud Configuration Optimization for Recurring Data-Intensive Applications

基于多臂赌博机的最优资源配置算法 - 背景介绍



- 寻找任务的最优资源配置
  - 对于重复任务特别重要，比如日常日志处理。
  - 糟糕的资源配置，会消耗5-6倍的完成时间，花费10倍以上的金钱。
- 挑战
  - 任务完成时间随着资源配置变化，难以预测。
  - 可计算云上实例价格随时间波动。

基于多臂赌博机的最优资源配置算法 - 基础模型



- Accordia: 动态寻找重复任务的最优资源配置。
- 寻找完成任务最便宜的资源配置，并且可以满足任务完成时间要求。

$$\min_{\boldsymbol{x}} g(\boldsymbol{x}) \cdot p_t(\boldsymbol{x}) \quad s.t.\ g(\boldsymbol{x}) \le T_{max} \qquad (P1)$$

- $g(\boldsymbol{x})$是资源配置$\boldsymbol{x}$ 下的任务完成时间。
- $p_t(\boldsymbol{x})$ 在 t时刻，单位时间的资源所需价格。
- T_{max} 最长任务完成时间。

基于多臂赌博机的最优资源配置算法 - 相关工作

- 有监督机器学习的寻找算法
  - Wrangler[1]， Paris[2] 使用SVM和随机森林算法，需要训练。
- 遗传算法和贝叶斯优化算法
  - LinkedIn[3]， Saboori et. al[4] 使用遗传算法解决黑箱优化问题。
  - CherryPick[5], Boat[6] 使用贝叶斯优化 EI算法，寻找最优资源。
- 需要预训练，或者需要长时间收敛，没有考虑实例价格随时间变化。

[1] Neeraja J Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. ACM SoCC 2014.
[2] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. 2017. Selecting the best vm across multiple public clouds: A data driven performance modeling approach. ACM SoCC 2017.
[3] Linkedin. 2016. Open Sourcing Dr. Elephant: Self-Serve Performance Tunning for Hadoop and Spark. https://github.com/linkedin/dr-elephant.
[4] Anooshiravan Saboori, Guofei Jiang, and Haifeng Chen. 2008. Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. IEEE ICDCS 2008.
[5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics.. In NSDI 2017.
[6] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. 2017. BOAT: Building auto-tuners with structured Bayesian optimization. In Proceedings of the 26th International Conference on World Wide Web. International World Wide Web Conferences Steering Committee, 479–488.

Related Work of Cloud Configuration Optimization Problem

Reference	Search Algorithm	Learning Mechanism	Handling Time-varying Price	Performance Guarantee
Wrangler [SoCC 2014]	Use cloud config. to predict running time via SVM	Offline pre-training	**No**	**No**
Paris [SoCC 2017]	Use cloud config. to predict running time via random forest.	Offline pre-training	**No**	**No**
Selecta [USENIX ATC 2018]	Use collaborative filtering across config. and running time to reduce the overhead.	Offline pre-training	**No**	**No**
LinkedIn [LinkedIn]	Use genetic algorithm to search the cloud config. with minimum running time	Online learning	**No**	**No**
Saboori et. Al [ICDCS 2008]	Use evolutionary algorithm to search the cloud config. with minimum completion cost	Online learning	**No**	**No**
CherryPick[NSDI 2017] *Boat* [WWW 2017] *Arrow* [ICDCS 2018]	Use expectation improvement (EI) algorithm to search the cloud config. with minimum completion cost in Bayesian optimization framework	Online learning	**No**	**No**
Accordia	Extend the Gaussian Process UCB algorithm to search the cloud config. with minimum completion cost	Online learning	**Yes**	$O(\sqrt{TlogT})$ **sub-linear regret**

- Accordia is online-learning based algorithm and can handle the time-varying price case.

Multi-armed Bandit Framework

- Problem setting
  - A gambler must decide which arm of N different slot machines to play in a sequence of trails so as to maximize his overall reward.
- Multi-armed bandit framework
  - Number of arms N, arm i has a fixed but unknown reward distribution P_i with expectation μ_i.
  - Number of rounds T, for $t = 1, 2, \ldots, T$:
    - The player chooses an arm i_t;
    - Arm i_t generates a random reward $X_{i,t} \in [0,1]$ drawing from P_i ;

基于多臂赌博机的最优资源配置算法 - 算法设计

- 转化为多臂赌博机问题(P2)

$$\min_{\boldsymbol{x}} g(\boldsymbol{x}) \cdot p_t(\boldsymbol{x}) \quad s.t.\ g(\boldsymbol{x}) \le T_{max} \ (P1) \xrightarrow[f(\boldsymbol{x}) \coloneqq 1/g(\boldsymbol{x})]{\text{inverse}} \max_{\boldsymbol{x}} f(\boldsymbol{x})/p_t(\boldsymbol{x}) \quad s.t.\ f(\boldsymbol{x}) \ge 1/T_{max} \ (P2)$$

  - $f(\boldsymbol{x})$ 和 $p_t(\boldsymbol{x})$ 未知，不能寻找到最优解 $\boldsymbol{x}_t^*$ 。
- 使用改进的 Gaussian-Process UCB 算法，求解 (P2)
  - 假设 $f(\boldsymbol{x})$ 服从高斯过程 $\mathcal{N}(\mu_t(\boldsymbol{x}_t), \sigma_t^2(\boldsymbol{x}_t))$.
  - 选择如下资源配置: $\max_{\boldsymbol{x}_t} \mu_t(\boldsymbol{x}_t)/\tilde{p}_t(\boldsymbol{x}_t) + \beta_t^{1/2}\sigma_t(\boldsymbol{x}_t)/\tilde{p}_t(\boldsymbol{x}_t)$

    $s.t.\ \ \mu_t(\boldsymbol{x}_t) - \beta_t^{1/2}\sigma_t(\boldsymbol{x}_t) \ge 1/T_{max}$
  - $\beta_t = 2\log(|X|t^2\pi^2\delta/6)$，$\delta \in (1, \infty)$. $\tilde{p}_t(\boldsymbol{x}_t)$ 是预测的实例价格。
- 评价标准：使用 regret 衡量表现 $Reg_T^S \coloneqq \sum_{t=1}^{T} f(\boldsymbol{x}^*)/p(\boldsymbol{x}^*) - \sum_{t=1}^{T} f(\boldsymbol{x}_t)/p(\boldsymbol{x}_t)$

可以证明 $\Pr\left\{ Reg_T^S \le \sqrt[2]{\log(\delta|X|)\, T(logT)^{d+1}} \right\} \ge 1 - 1/\delta$

Implement Accordia for Spark over Kubernetes

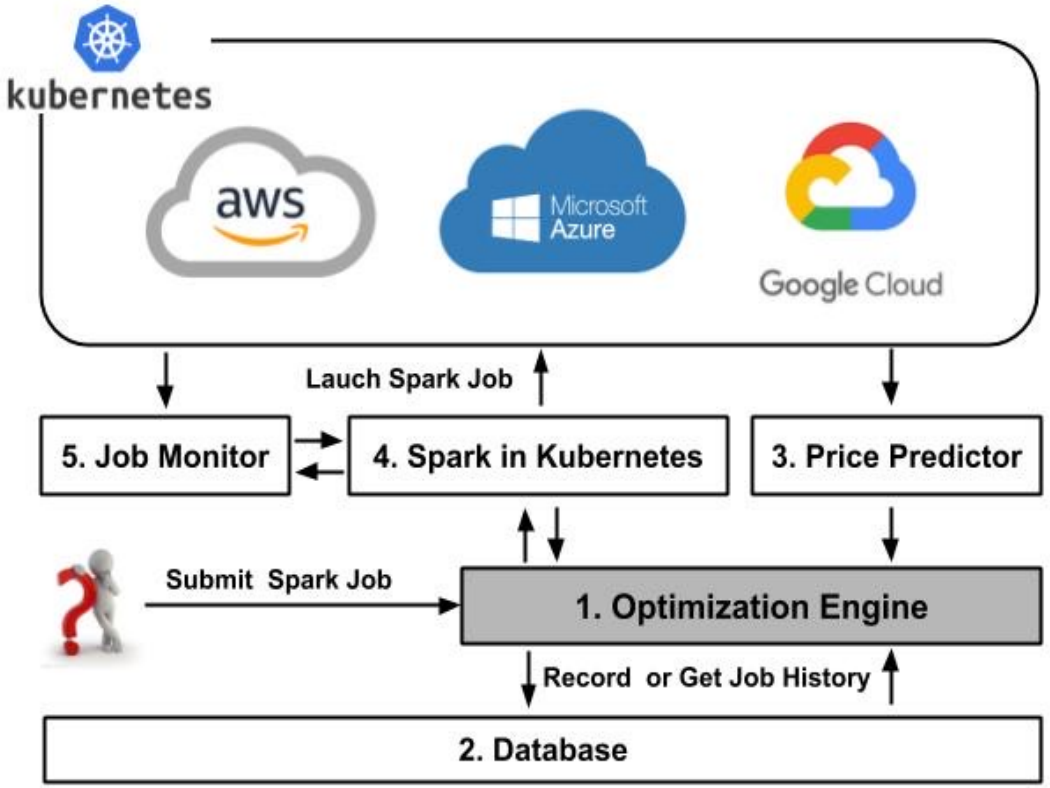


Figure 1. System architecture of Accordia for Spark applications over Kubernetes

- Use Accordia to optimize:
  - The no. of Spark executors.
  - The no. of CPU cores and Memory (RAM) size for the driver and executor pods
  - A 5-dimensional search space with 7000 candidate cloud config..
- Four components:
  - **Database** stores history information and has a sliding window to focus on recent history.
  - **Price Predictor** collects the real-time price and predict the future price.
  - **Optimization Engine** implements Accordia algorithm to select the next-run cloud config..
  - **Job Monitor** aborts a partially-run job with 30% larger completion cost than the best one.

Insight of Gaussian Process UCB Algorithm

- Gaussian Process model
  - Job completion time $f(\boldsymbol{x})$ is highly correlated with cloud configuration $\boldsymbol{x}$.
- Upper Confidence Bound (UCB) algorithm:
  - Balance exploitation $\mu_t(\boldsymbol{x}_t)/\widetilde{p}_t(\boldsymbol{x}_t)$ and exploration $\sigma_t(\boldsymbol{x}_t)/\widetilde{p}_t(\boldsymbol{x}_t)$.
- Spark WordCount example

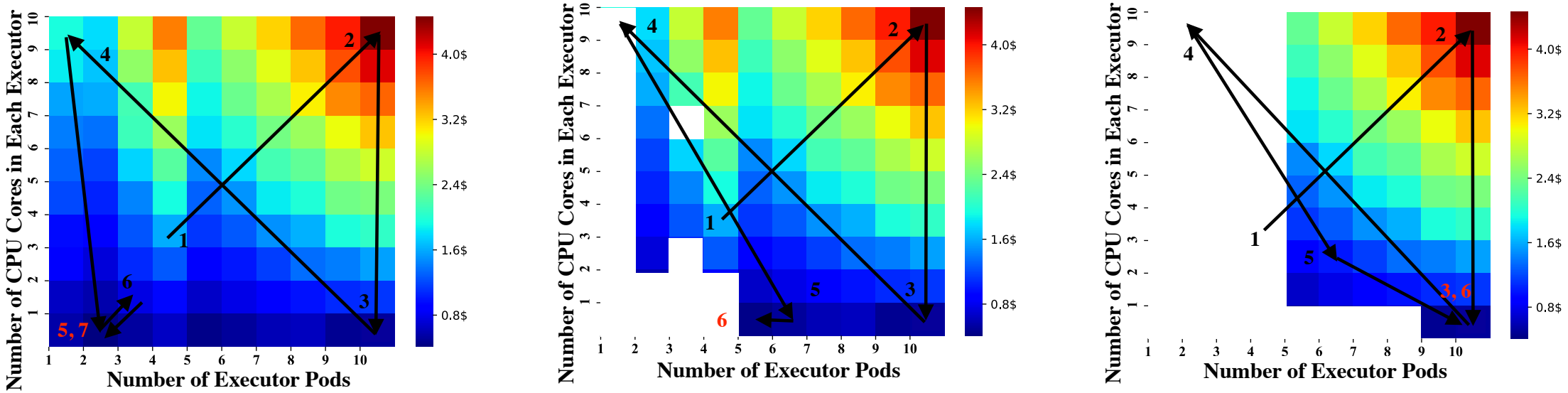


Figure 1. Sequence of cloud configurations chosen by Accordia with/ without a deadline constraint.

Evaluate Accordia Performance in Public Clouds

- Figure 1 and Figure 2 show Accordia can find the near-optimal configuration among 7000 candidate configurations within 20 runs, 2X-speedup and 20.9% cost-savings comparing to CherryPick.
- Accordia submits one recurring job every day and dynamic switches job types without notifying the system every 30 days in Figure 3.
- Accordia still can obtain up to 18.6% cost-savings under dynamic switching job types over exponentially-distributed time-intervals (i.e., Poisson distribution with $\lambda = 20$).

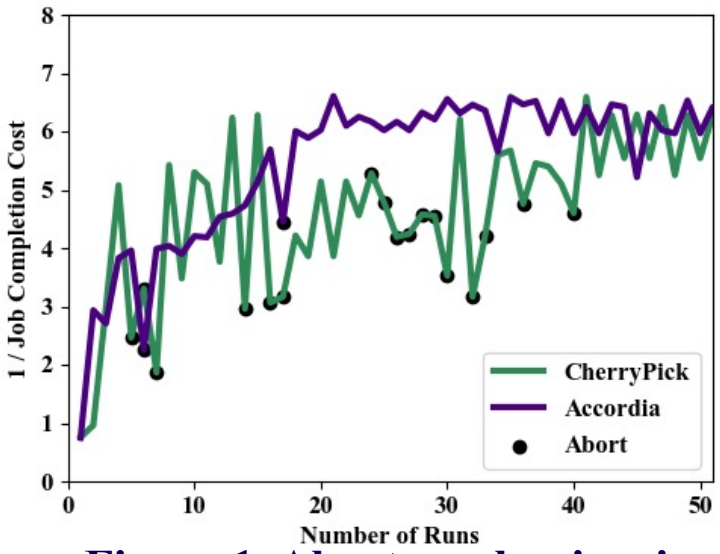


Figure 1. Abort mechanism in SparkPi example.

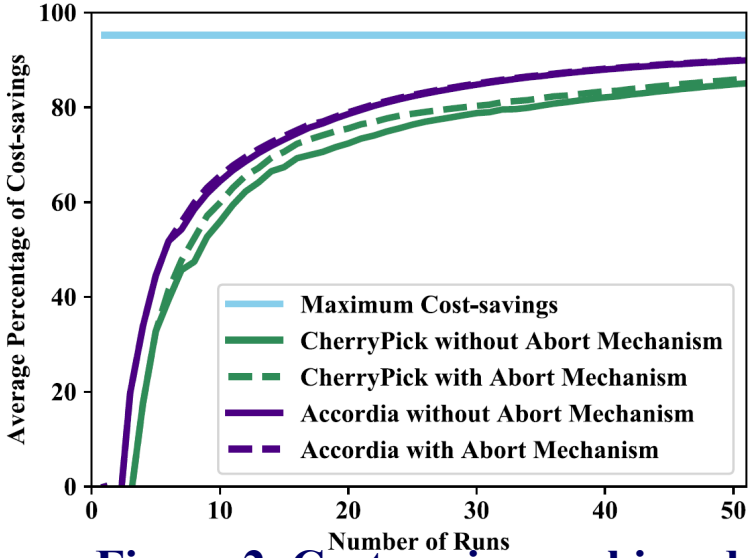


Figure 2. Cost-savings achieved by Accordia.

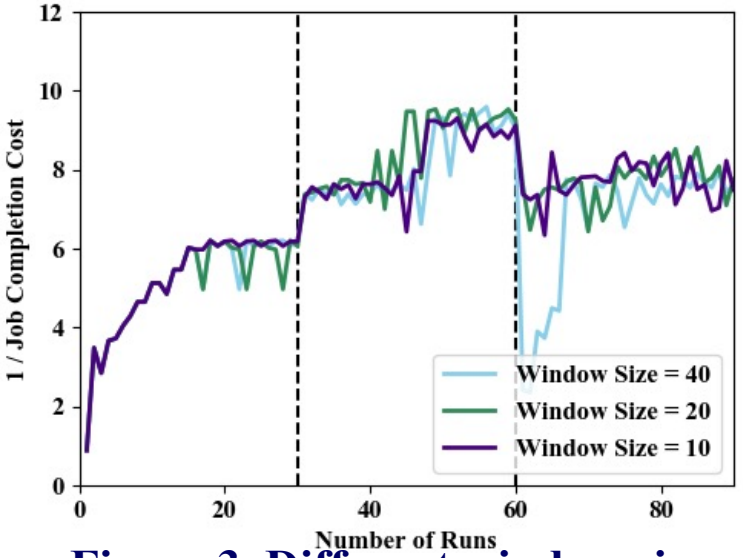


Figure 3. Different window sizes under dynamic switching job types.

Dragster: Online Resource Optimization for Elastic Stream Processing with Regret Guarantee

Introduction of Distributed Stream-processing Systems

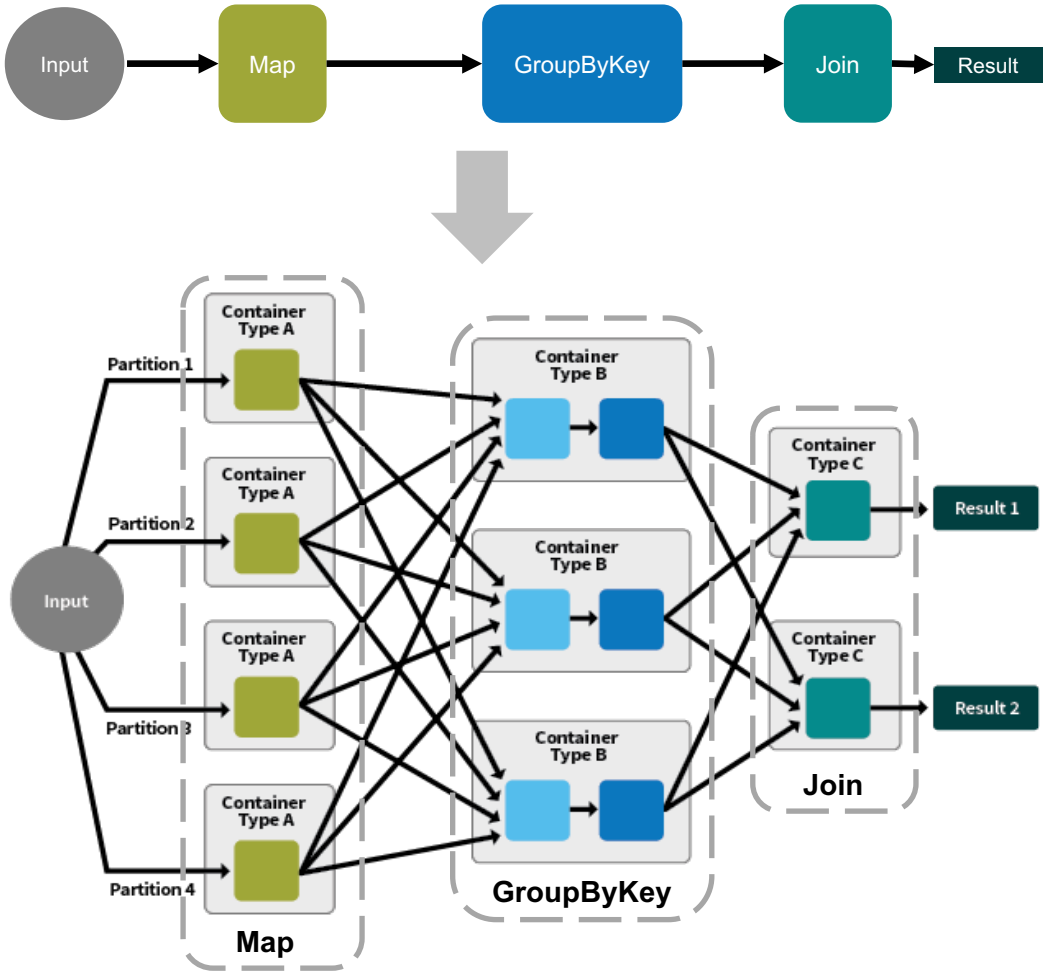


Figure 1. A data stream graph example.

- 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

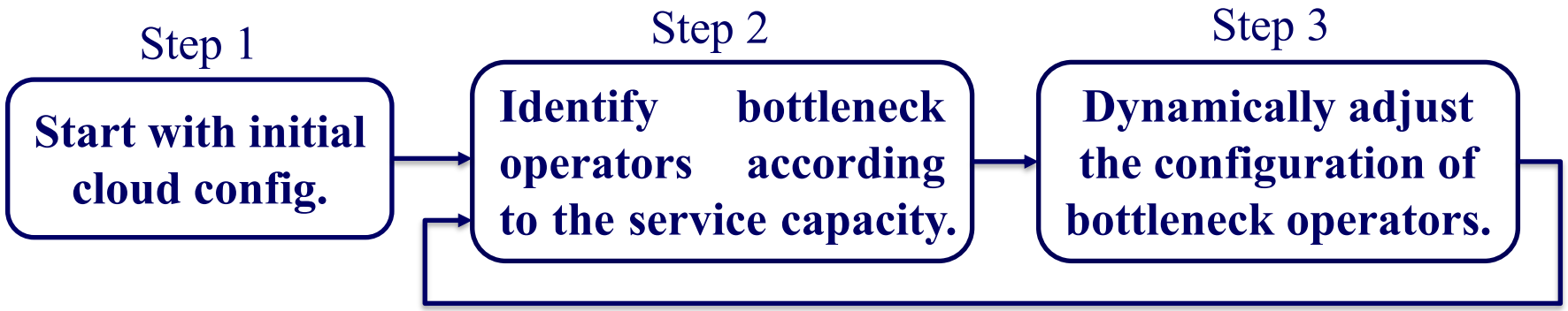


Figure 1. Dragster Workflow

- 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.
- Two-level online learning approach:
  - Model-driven approach: identify the bottleneck operator according to the service capacity.
  - Data-driven approach: Adjust the configuration of bottleneck operator.

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..

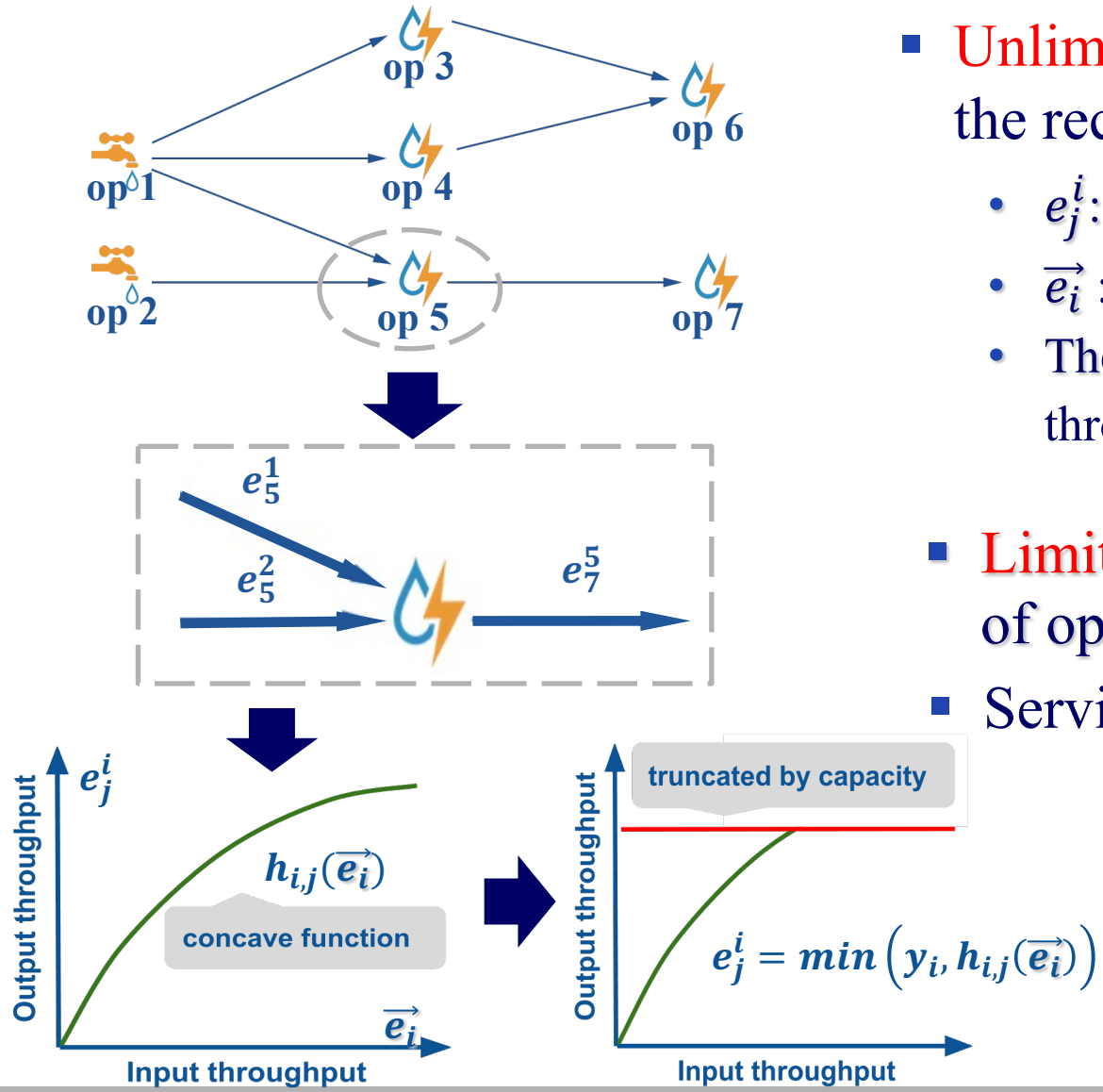
Related Work of Cloud Configuration Optimization Problem

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."

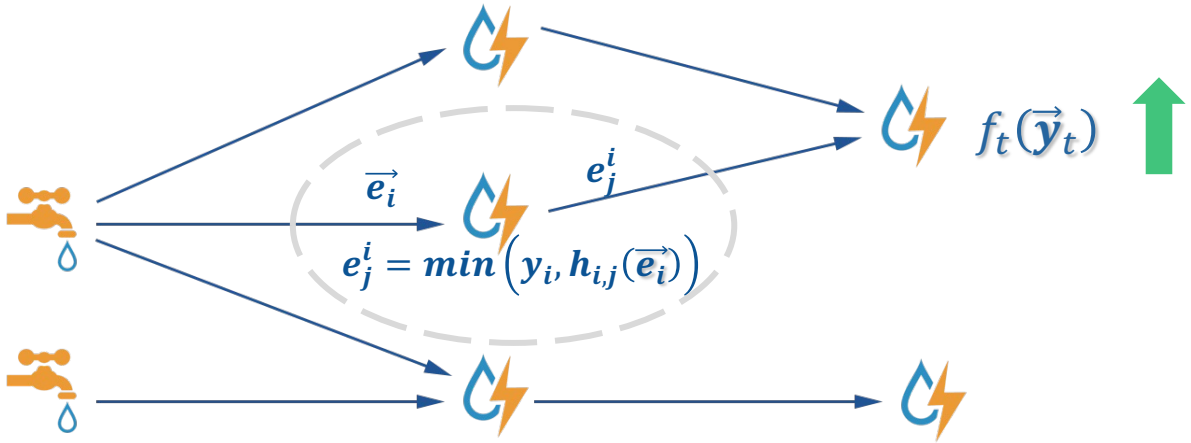
Level 1: Identify the Bottleneck Operator – Operator Wise



- Unlimited service capacity: operator i can process all the received tuples.
  - e_j^i: throughput from operator i to operator j.
  - $\vec{e_i}$: the received throughput vector.
  - The throughput function $h_{i,j}$ captures the input and output throughput mapping under unlimited service capacity case.
- Limited service capacity y_i: maximum processing rate of operator i.
- Service capacity y_i truncates the throughput function $h_{i,j}$:

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

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 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.

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

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

Go Back to Accordia

Two-Level Online Optimization Framework – Summary

Level 1: Identify the bottleneck operator

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

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

→ Goal: maximize the accumulated throughput.

→ Long-term constraint: operator buffer constraint.

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

- 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

---

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.

- 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

Implement Dragster for Apache Flink over Kubernetes

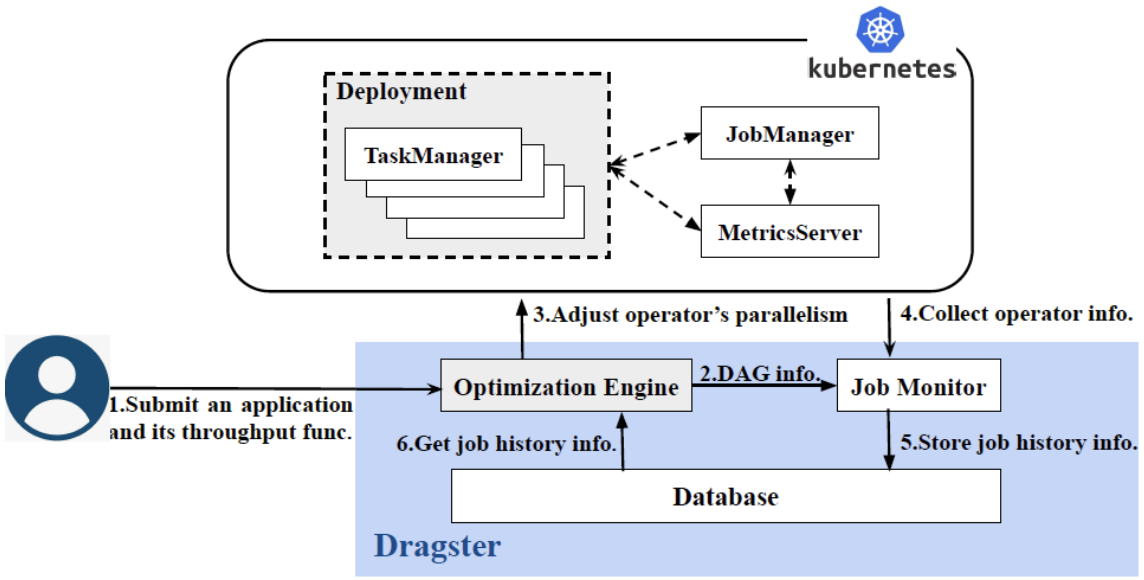


Figure 1. System architecture of Dragster for Apache Flink applications over Kubernetes

- 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.

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.

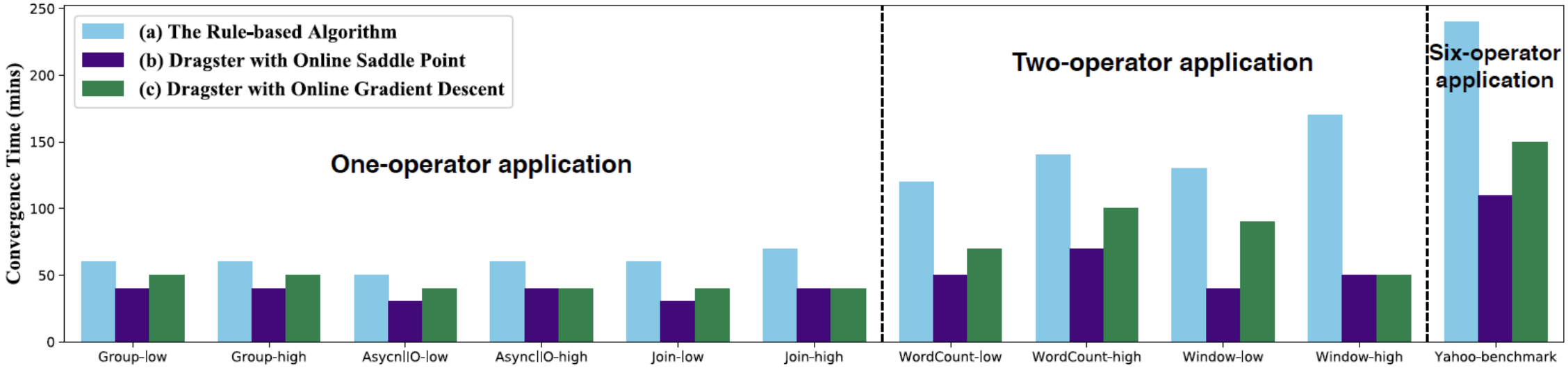


Figure 1. Convergence time under a diverse set of workloads.

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.

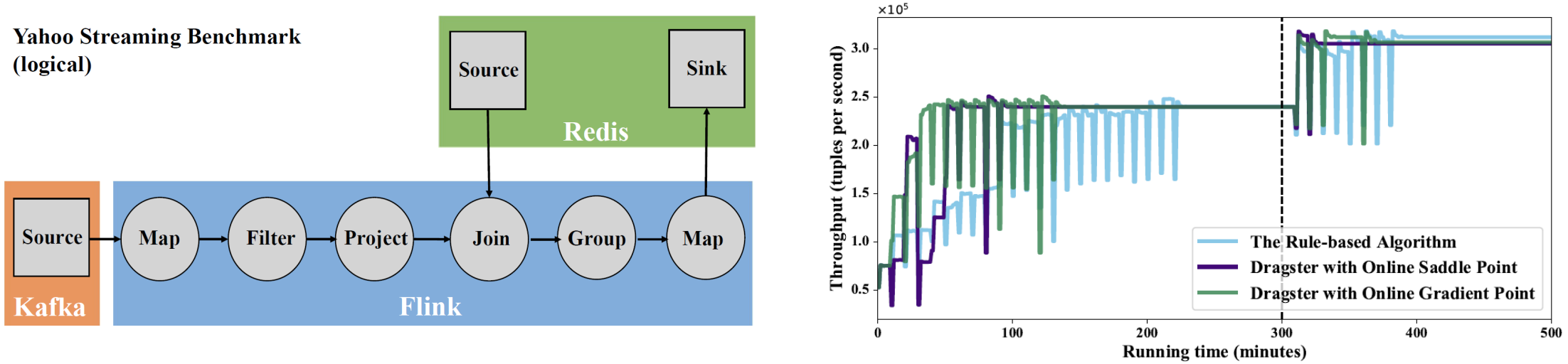


Figure 1. Throughput achieved by Dragster under workload changing running Yahoo streaming benchmark.

DRAM: API-aware Dynamic Resource Allocation for Microservices

Introduction of Microservice(MS) architecture

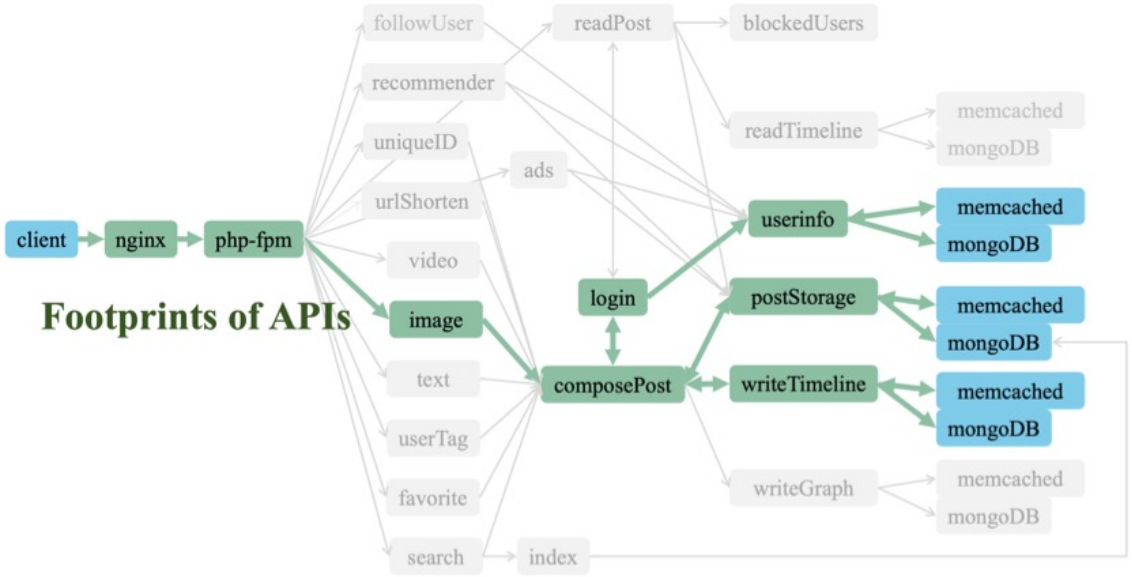
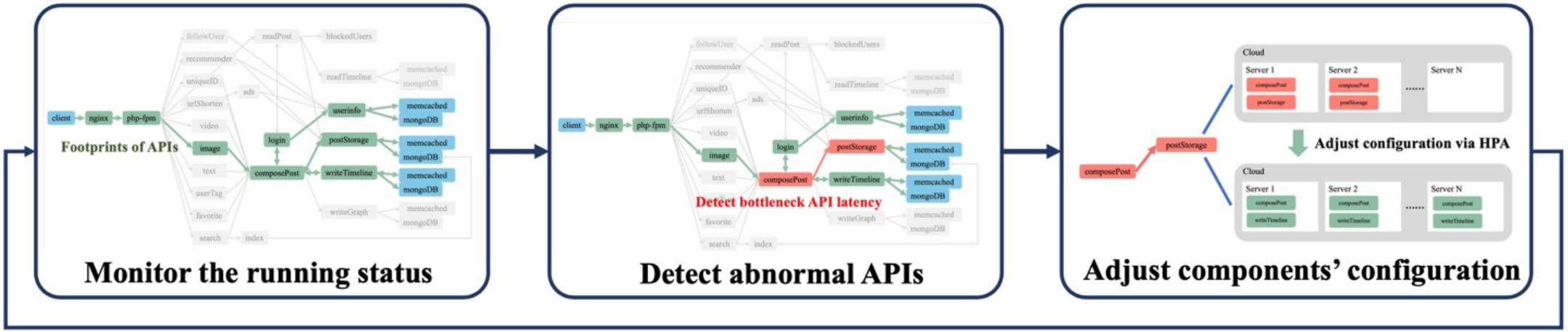


Figure 1. MS Call graph of Social Network.

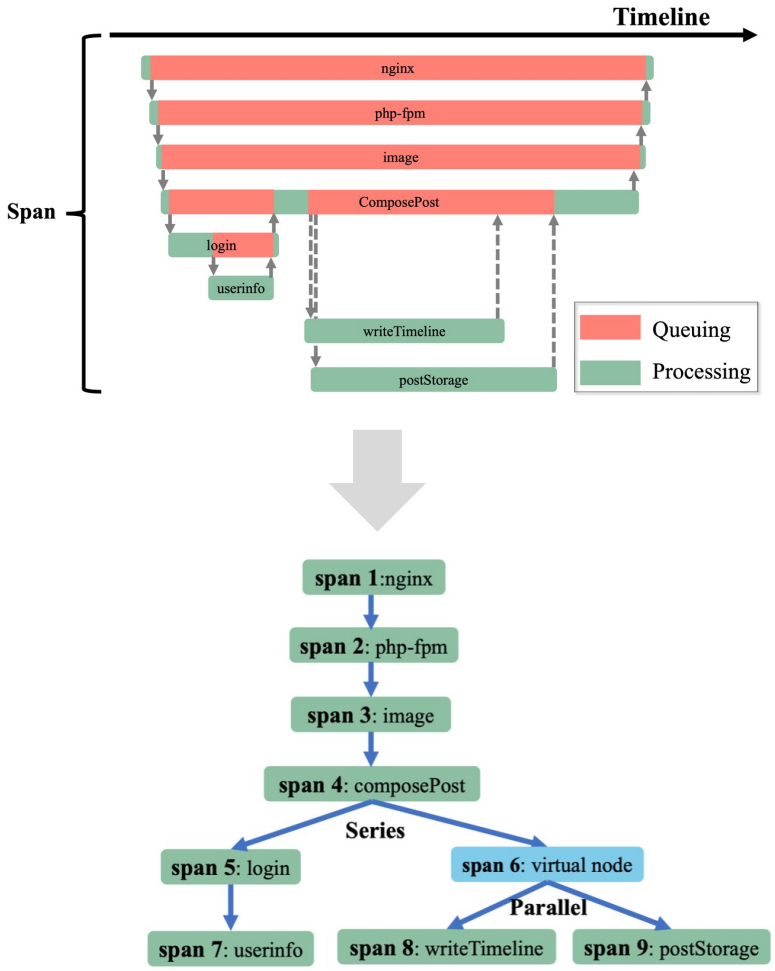
- Microservice application process incoming requests via a MS call graph.
- Node is a MS component which can run on multiple containers in parallel.
- Edge indicates how request invokes the downstream components.
- A request invokes several paths of components across the MS call graph.

API-aware Config. Optimization for Microservice



- ADRAM: dynamically adjust configuration for each MS component to minimize the number of SLA violations for critical requests with varying workloads under a budget constraint.
- Challenge compared to streaming processing system:
  - MS call graph is very large, hundreds of components, millions of containers, billions of requests.
  - Focus on critical requests (e.g., focus on e-payment request NOT log processing).
  - Request invokes downstream components in parallel/ series (can not capture in MS call graph).

Level 1: Identify the Bottleneck Component – Component Wise



- Use span graph in distributed tracing system to capture the temporal order of invocations.
- The duration d_i of component i can be separated as queuing time q_i + processing time p_i .

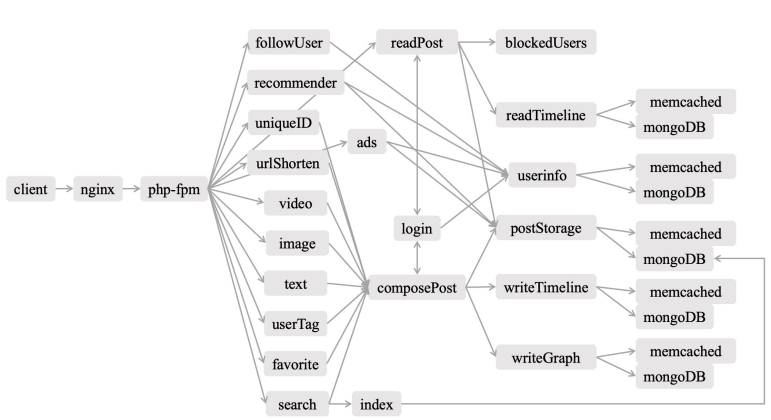
$$d_i = \mathrm{q_i} + \mathrm{p_i}$$

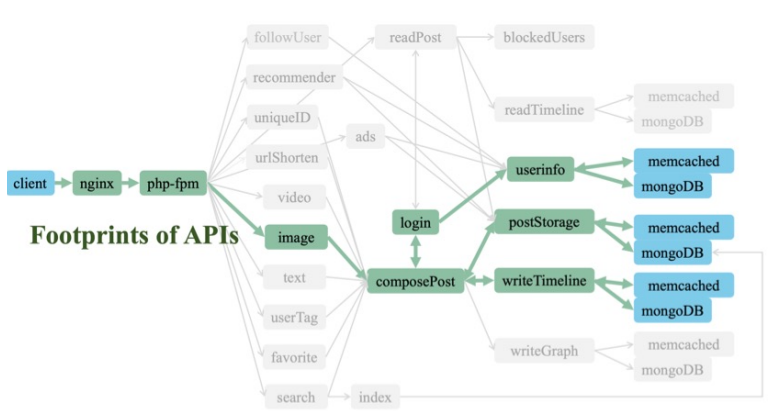
- The queuing time q_i results from invocations of downstream components in parallel / series.

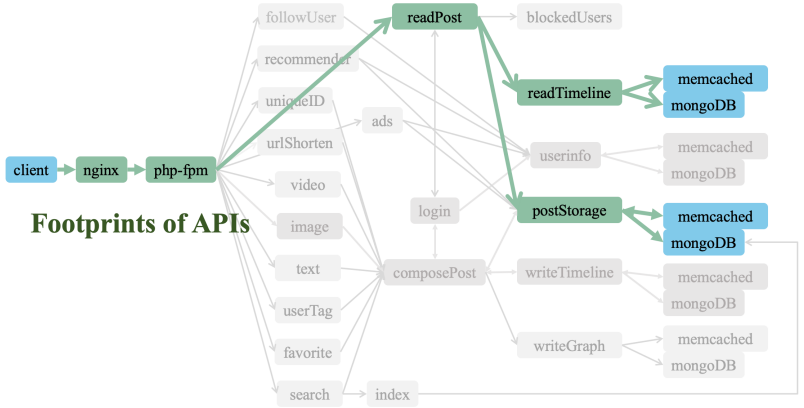
$$q_i = \max_{j \in DS_i}\{d_j\} \ / \ q_i = \sum_{j \in DS_i} d_j$$

- If the downstream invocations are in mixture parallel and series, then virtual components can fix it.

Level 1: Identify the Bottleneck Component – Request Wise







- The latency of a request k is the duration of root component , represent by $f_k(\vec{\boldsymbol{p}}_t)$, where $\vec{\boldsymbol{p}}_t$ is the processing time vector.
- Object: minimize the accumulated SLA violations.

$$\min_{\vec{\boldsymbol{p}}_t} \ \sum_k a_k \, r_k \ \cdot \max\{0, f_k(\vec{\boldsymbol{p}}_t) \ - SLA_k\}$$

where a_k is the weight, r_k is the request arriving rate.

- The solution is the target processing time for each component.

Level 2: Adjust the Config. of Bottleneck Operator

- The processing time p_i can not be set directly since it depends on its resource configuration $\boldsymbol{x}_i$ and offered workloads $\boldsymbol{z}_i$ in an unknown manner.
- Take an online learning-based technique, contextual Gaussian-Process multi-armed bandit.
- Assume the processing time p_i following a Gaussian Process model.

$$p_i \sim GP(\mu_{i,t}(\boldsymbol{x}_i(t), z_i(t)), \sigma^2_{i,t}(\boldsymbol{x_i}(t), z_i(t)))$$

- Update the posterior distribution to capture the mapping between configuration and processing rate.

Two-Level Online Optimization Framework – Summary

Level 1: Identify the bottleneck component

$$\min_{\vec{p}_t} \sum_k a_k r_k \cdot \max\{0, f_k(\vec{p}_t) - SLA_k)\}$$ → Goal: minimize the accumulated SLA violations.

- Identify the bottleneck operator according to the processing time
  - Model-based techniques: the end-to-end latency depends on the critical path
  - Model-based techniques: requests compete for resources

Level 2: Adjust the configuration of the bottleneck component

$$p_i \sim GP(\mu_{i,t}(\boldsymbol{x}_i(t), z_i(t)), \sigma^2_{i,t}(\boldsymbol{x}_i(t), z_i(t)))$$ → The service capacity follows a GP model.

$$\sum_i \boldsymbol{x}_i \leq B$$ → Resource budget constraint.

- Adjust the configuration of bottleneck operator
  - Allocate just enough computing resources to obtain the target processing time, no wasting
  - Learning-based techniques: formulate into a contextual multi-armed bandit problem

Go Back to Dragster