Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing

Chenghao Lyu†, Qi Fan‡, Fei Song‡, Arnab Sinha‡, Yanlei Diao†‡, Wei Chen‡, Li Ma‡, Yihui Feng‡, Yaliang Li*, Kai Zeng‡, Jingren Zhou*

† University of Massachusetts, Amherst; ‡ Ecole Polytechnique; * Alibaba Group
chenghao@cs.umass.edu, {qi.fan,fei.song,arnab.sinha,yanlei.diao}@polytechnique.edu
{wickeychen.cw,mali.mali,yihui.feng,yaliang.li,zengkai.zk,jingren.zhou}@alibaba-inc.com

ABSTRACT

Big data processing at the production scale presents a highly complex environment for resource optimization (RO), a problem crucial for meeting performance goals and budgetary constraints of analytical users. The RO problem is challenging because it involves a set of decisions (the partition count, placement of parallel instances on machines, and resource allocation to each instance), requires multi-objective optimization (MOO), and is compounded by the scale and complexity of big data systems while having to meet stringent time constraints for scheduling. This paper presents a MaxCompute based integrated system to support multi-objective resource optimization via fine-grained instance-level modeling and optimization. We propose a new architecture that breaks RO into a series of simpler problems, new fine-grained predictive models, and novel optimization methods that exploit these models to make effective instance-level RO decisions well under a second. Evaluation using production workloads shows that our new RO system could reduce 37-72% latency and 43-78% cost at the same time, compared to the current optimizer and scheduler, while running in 0.02-0.23s.

1 INTRODUCTION

While big data query processing has become commonplace in enterprise businesses and many platforms have been developed for this purpose [3, 5, 6, 8, 11, 26, 29, 39, 46, 53, 57, 61, 62], resource optimization in large clusters [13, 32–34] has received less attention. However, we observe from real-world experiences of running large compute clusters in the Alibaba Cloud that resource management plays a vital role in meeting both performance goals and budgetary constraints of internal and external analytical users.

Production-scale big data processing presents a highly complex environment for resource optimization. We show an example through the life cycle of a submitted job in MaxCompute [26], the big data query processing system at Alibaba, as shown in Fig. 1. A submitted SQL job is first processed by a “Cost-Based Optimizer” (CBO) to construct a query plan in the form of a Directed Acyclic Graph (DAG) of operators. These operators are further grouped into several stages connected with data shuffle (exchanging) dependencies. As shown in Fig. 1, job 1 is composed of stage 1 (operators 1-3), stage 2 (operators 4-8), and stage 3 (operators 9-15), and their boundaries are data shuffling operations. To explore data parallelism, each stage runs over multiple machines and each machine runs an instance of a stage over a partition of the input data. To enable such parallel execution, a “History-Based Optimizer” (HBO) recommends a partition count (number of instances) for each stage and a resource plan (the number of cores and memory needed) for all instances of the stage based on previous experiences.

During job execution, a Stage Dependency Manager will maintain the stage dependencies of a job and pop out all the unblocked stages to the Fuxi scheduler [61]. The scheduler treats each stage as a task to be scheduled and maintains tasks in queues. For each stage, the Fuxi scheduler uses a heuristic-based approach to recommending a placement plan that sends instances to machines. After an instance is assigned to a machine, it will be executed using the resource plan that HBO has created for this instance, which is the same for all instances of the same stage.

Challenges. MaxCompute’s large, complex big data processing environment poses a number of challenges to resource optimization. First, resource optimization involves a set of decisions that need to be made in the life cycle of a big data query: (1) the partition count of a stage; (2) the placement plan that maps the instances of a stage to the available machines; (3) the resource plan that determines the resources assigned to each instance on a given machine. All of these factors will have a significant impact on the performance of the job, e.g., its latency and computing cost. Among the three issues, the partition count is best studied in the literature [13, 16, 32, 33]. But this decision alone is not enough – both decisions 2 and 3 can affect the performance a lot. While most of the existing work neglects the placement problem, due to the use of virtualization or container technology, a service provider like Alibaba Cloud does need to solve the placement problem in the physical clusters. As for the resource plan, it is determined by HBO from past experiences, without considering the latencies of the current instances in hand. However, such solutions are far from ideal: if one underestimates the resource needs, the job can miss its deadline. On the other hand, overestimation leads to wasted resources and higher costs on (both internal and external) users. A few systems addressed the placement problem [13, 16] or the resource problem [34, 44] in isolation and in simpler system environments. We are not aware of any solution that can address all three problems in an integrated system.

Second, the scale and complexity of our production clusters make the above decisions a challenging problem. Most notably, our clusters can easily extend to tens of thousands of machines, while all the resource optimization decisions must be made well under a second. An algorithm for the placement problem with quadratic complexity [16] may work for a small cluster of 10 machines and a small number of jobs, as in the original paper, but not for the scale of 10’s thousands of machines and millions of jobs. Furthermore, the optimal solutions to both the placement and resource plans depend on the workload characteristics of instances, available machines,
and current system states. It is nontrivial to characterize all the data needed for resource optimization, let alone the question of using all the data to derive optimal solutions well under a second.

Third, our use cases clearly indicate that resource optimization is a multi-objective optimization (MOO) problem. A real example we encountered in production is that after the user changed to allocate 10x more resources (hence paying 10x the cost), the latency of a job was reduced only by half, which indicates ineffective use of resources and a poor tradeoff between latency and cost. Over such a complex processing system, the user has no insights into how latency and cost trade-off. Therefore, there is a growing demand that the resource optimizer makes the decisions automatically to achieve the best tradeoff between multiple, often competing, objectives. Most existing work on resource optimization focuses on a single objective, i.e., job latency [15, 16, 17, 32, 33, 44, 59]. Only a few recent systems [34, 38] offer a multi-objective approach to the resource optimization problem. Still, their solutions do not suit the complex structure of big data queries, which we detail below.

Fourth, to enable multi-objective resource optimization, the system must have a model for each objective to predict its value under any possible solution that the optimizer would consider. Existing models, derived from domain knowledge [15, 16, 32] or machine learning methods [21, 24, 35, 63], have been used to improve SQL query plans [23, 43] or to improve selectivity estimation [10, 18, 19, 30, 31, 36, 48–52, 54, 55, 64]. But a key observation made in this work is that existing models do not suit the needs of resource optimization of big data queries because they perform only coarse-grained modeling: by capturing only end-to-end query latency or operator latency across parallel instances, these models may yield highly variable performance as they often involve large numbers of stages and parallel instances. Running optimization on highly-variable models gives undesirable results while missing opportunities for instance-level recommendations.

Example 1. Fig. 2(a) and 2(b) show that in a production trace of 0.62 million jobs, there are 1.9 million stages in total, with up to 64 stages in each job, and 121 million instances, with up to 81430 instances in a stage. For a particular stage with 6716 instances, Fig. 2(c) shows that the latencies of different instances vary a lot. If a performance model captures only the overall stage latency, i.e., the maximum instance latency, when the resource optimizer is asked to reduce latency, it will assign more resources uniformly to all the instances (as they are not distinguishable by the model). The extra resources do not contribute to the stage latency for those short-running instances, while incurring a higher cost. Instead, an optimal solution would be only to assign more resources to long-running instances while reducing resources for short-running ones. Such decisions require fine-grained instance-level models as well as instance-specific resource plans.

**Contributions.** Based on the above discussion, our work aims to support **multi-objective resource optimization via fine-grained instance-level modeling and optimization**, and devises a new system architecture and algorithms to enable **fast resource optimization decisions, well under a second, in the face of 10’s thousands of machines and 10’s thousands of instances per stage**. By way of addressing the above challenges, our work makes the following contributions.

1. **New architecture of a resource optimizer** (Section 3). To enable all the resource optimization (RO) decisions of a stage within a second, we propose a new architecture that breaks RO into a series of simpler problems. The solutions to these problems leverage existing cost-based optimization (CBO) and history-based optimization (HBO), while fixing their suboptimal decisions using a new Intelligent Placement Advisor (IPA) and Resource Assignment Advisor (RAA), both of which exploit fine-grained predictive models to enable effective instance-level recommendations.

2. **Fine-grained models** (Section 4). To suit the complexity of our big data system, our fine-grained instance-level models capture all relevant aspects from the outputs of CBO and HBO, the hardware, and machine states, and embed these heterogenous channels of information in a number of deep neural network architectures.

3. **Optimizing placement and resource plans** (Section 5). We design a new stage optimizer that employs a new IPA module to derive a placement plan to reduce the stage latency, and a novel RAA model to derive instance-specific resource plans to further reduce latency and cost in a hierarchical MOO framework. Both methods are proven to achieve optimality with respect to their own set of variables, and are optimized to run well under a second.

4. **Evaluation** (Section 6). Using production workloads of 0.6M jobs and 1.9M stages and a simulator of the extended MaxCompute environment, our evaluation shows promising results: (1) Our best
model achieved 7-15% median error and 9-19% weighted mean absolute percentage error. (2) Compared to the HBO and Fuxi scheduler, IPA reduced the latency by 10-44% and the cloud cost by 3-12% while running in 0.04s. (3) IPA + RAA further achieved the reduction of 37-72% latency and 43-78% cost while running in 0.02-0.23s.

2 RELATED WORK

ML-based query performance prediction. Recent work has developed various Machine Learning (ML) methods to predict query performance. QPPNet [24] builds separate neural network models (neural units) for individual query operators and constructs more extensive neural networks based on the query plan structure. Each neural unit learns the latency of the subquery rooted in a given operator. TLSTM [35] constructs a uniform feature representation for each query operator and feeds the operator representations into a TreeLSTM to learn the query latency. Both approaches, however, are tailored only for a single machine with an isolated runtime environment and fixed resources. Hence, they are not directly applicable to our RO problem in large-scale complex clusters.

ModelBot2 [21] trains ML models for fine-grained operating units decomposed from a DBMS architecture to enable a self-driving DBMS. However, it is designed for a local DMBS but not big data systems as in our work. GPredictor [63] predicts the latency of concurrent queries in a single machine with a graph-embedding-based model. It further improves accuracy by using profiling features such as data-sharing, data-conflict and resource competition from the local DBMS. In our work, however, concurrency information regarding the operators from different queries is unavailable due to the container technology in large clusters [26, 46, 57, 62].

ML-based models have been used for different purposes. NEO [23] learns a DNN-based cost model for (sub)queries and uses it to build a value-based Reinforcement Learning (RL) algorithm for improving query execution plans. Vaidya et al. [43] train ML models from query logs to improve query plans for parametric queries. Phoebe [65] uses ML models for improving checkpointing decisions. Many recent works [10, 18, 19, 30, 31, 36, 48-52, 54, 55, 64] have applied ML-based approaches to improve cardinality estimation.

A final, yet important, comment on the above systems is that they do not address the RO problem like in our work.

Performance tuning in DBMS and big data systems. Performance tuning systems require a dedicated, often iterative, tuning session for each workload, which can take long to run (e.g., 15-45 minutes [44, 59]). As such, they are not designed for production workloads that need to be executed on demand. In addition, they aim to optimize a single objective, e.g., query latency. Among search-based methods, BestConfig [67] searches for good configurations by dividing high-dimensional configuration space into subspaces based on samples, but it cold-starts each tuning request. ClassyTune [66] solves the optimization problem by classification, which cannot be easily extended to the MOO setting. Among learning-based methods, Ottertune [44] builds a predictive model for each query by leveraging similarities to past queries, and runs Gaussian Process exploration to try other configurations to reduce latency. CDBTune [59] uses Deep RL to predict the reward (a weighted sum of latency and throughput) of a given configuration and explores a series of configurations to optimize the reward. ResTune [60] uses a meta-learning model to learn the accumulated knowledge from historical workloads to accelerate the tuning process for optimizing resource utilization without violating SLA constraints.

Task scheduling in big data systems. Fuxi [61] and Yarn [46] make the scheduling decisions based on locality information, while Trident [11] improves Yarn by considering the locality in different storage tiers. However, these systems treat each task as a blackbox and make scheduling decisions without knowing the task characteristics, such as the query plan structures and performance predictions. Hence, they cannot find optimal solutions to the machine placement and/or resource allocation problems.

Resource optimization in big data systems. In cluster computing, a resource optimizer (RO) determines the optimal resource configuration on demand and with low latency as jobs are submitted. RO for parallel databases [16] determines the best data partitioning strategy across different machines to minimize a single objective, latency. Its time complexity of solving the placement problem is quadratic to the number of machines (9 machines in [16]), which is not affordable on today’s productive scale (>10K machines). Morpheus [13] codifies user expectations as SLOs and enforces them using scheduling methods. However, its optimization focuses on system utilization and predictability, but not minimizing the cost and latency of individual jobs as in our work. PerOrator [32] solves a single-objective (latency) optimization problem via an exhaustive search of the solution space while calling its model for predicting the performance of each solution. WiseDB [22] manages cloud resources based on a decision tree trained on performance and cost features from minimum-cost schedules of sample workloads, while such schedules are not available in our case. Li et al. [17] minimize end-to-end tuple processing time using deep RL and requires defining scheduling actions and the associated reward, which is not available in our problem. Recent work [15] proposes a heuristic-based model to recommend a cloud instance (e.g., EC2 instance) that achieves cost optimality for OLAP queries, which is different from our resource optimization problem.

CLEO [33, 47] learns the end-to-end latency model of query operators, and based on the model, minimizes the latency of a stage (involving multiple operators) by tuning the partition count. It has a set of limitations: First, its modeling target concerns the latency of multiple instances over different machines, which can be highly variable (e.g., due to uneven data partitions or scheduling delays) and hard to predict. Second, CLEO’s latency model does not permit instance-level recommendations for the placement and resource allocation problems. Third, its optimization supports a single objective, and determines only the partition count in a stage, but not other RO decisions. Bag et al. [2] propose a plan-aware resource allocation approach to save resource usage without impacting the job latency but not minimize latency and cost like in our work.

UDAQ [34, 58] tunes Spark configurations to optimize for multiple objectives. However, it works only on the granularity of an entire query and neglects its internal structure. Such coarse-grained modeling of latency is unlikely to be accurate for complex big data queries, which leads to poor results of optimization. TEMPO [38] considers multiple Service-Level Objectives (SLOs) of SQL queries and guarantees max-min fairness when they cannot be all met. But its MOO works for entire queries, but not fine-grained MOO that suits the complex structure of big data systems.
Multi-objective optimization (MOO). MOO for SQL [12, 14, 40–42] finds Pareto-optimal query plans by efficiently searching through a large set of them. The problem is fundamentally different from RO, including machine placement and resource tuning problems. MOO for workflow scheduling [14] assigns operators to containers to minimize total running time and money cost. But its method is limited to searching through 20 possible numbers of containers and solving a constrained optimization for each option.

Theoretical MOO solutions suffer from a range of performance issues when used in a RO: Weighted Sum [25] is known to have poor coverage of the Pareto frontier [27]. Normalized Constraints [28] lacks in efficiency due to repeated recomputation to return more solutions. Evolutionary Methods [7] approximately compute a Pareto set but suffer from inconsistent solutions. Multi-objective Bayesian Optimization explores the potential Pareto-optimal points iteratively by extending the Bayesian approach, but lacks the efficiency required by RO due to a long-running time. Progressive frontier [34] is the first MOO solution suitable for RO, offering good coverage, efficiency, and consistency. However, none of them consider complex structures of big data queries and require modeling end-to-end query latency, which does not permit instance-level optimization.

3 SYSTEM OVERVIEW

In this section, we provide the background on MaxCompute [26], the big data query processing system at Alibaba, and our proposed extended architecture for resource optimization.

3.1 Background on MaxCompute

In MaxCompute, a submitted user job is represented hierarchically using stages and operators, which will then be executed by parallel instances. As shown in Fig. 1, a job is a Directed Acyclic Graph (DAG) of stages, where the edges between stages are inter-machine data exchange (shuffling) operations. A stage is a DAG of operators, where edges are intra-machine pipelines without data shuffling. The input data of each stage is partitioned over different machines, where each partition is run as an instance of the stage in a container.

Fig. 1 shows the functionality of query optimizers and the scheduler in the lifecycle of a submitted job.

- Cost-Based Optimizer (CBO): MaxCompute’s Cost-Based Optimizer (CBO) follows traditional SQL optimization based on cardinality and cost estimation and generates a Physical Operator Tree (POT), which is a DAG of stages where each stage is a DAG of operators.
- History-Based Optimizer (HBO): To facilitate resource optimization, a History-Based Optimizer (HBO) gives an initial attempt to recommend a partition count (number of instances) for each stage and a resource plan (the number of cores and memory needed) for all instances of the stage based on previous experiences. This history-based approach is known to be suboptimal because it does not consider the machines to run these queries and their current states. Both hardware characteristics and system states affect the latencies of instances, and an optimal solution should try to minimize the maximum latency of parallel instances, in addition to cost objectives. Moreover, HBO needs expensive engineering efforts, especially when workloads change and the system upgrades. We will address these issues in our new system design.

Scheduler: During job execution, the granularity of scheduling is a stage: a stage that has all dependencies met is handed over to the Fuxi scheduler [61]. Fuxi uses a heuristic approach to recommending a placement plan that sends instances to machines, and each instance is assigned to a container on a machine with a previously-determined resource plan for CPU and memory. These decisions, however, are made without being aware of the latency of each instance. As such, an instance with a potential longer running time (e.g., due to larger input size) may be sent to a heavily loaded machine while another instance with less data may be sent to an idle machine, leading to overall poor stage latency (the maximum of instance latencies). A detailed example is given later in Fig. 6.

Workload and Cluster complexity. Production clusters take workloads with a vast variety of characteristics. Workload A from an internal department includes 1 - 64 stages in each job. A stage may involve 2 - 249 operators and be executed by 1 - 42K instances, where an instance could take sub-seconds up to 1.4 hours to run. Further, production clusters consist of heterogeneous machines. For example, we observed 5 different hardware types when executing workload A, where each hardware type includes 30 - 7K machines. Moreover, the system states in each machine vary over time. Take CPU utilization for example. The average CPU utilization varies from 32%-83%, and the standard deviation ranges from 6%-23%. Finally, each machine could run multiple containers, and a container runs an instance with a quota of CPU and memory based on a resource plan. For workload A, we observed 17 different resource plans for containers. Since there is no perfect isolation between containers and the host OS [45], containers with the same resource plan could perform differently on machines with different hardware or system states, making the running environment more complex.

3.2 System Design for Resource Optimization

We next show our design for resource optimization by extending the MaxCompute architecture. Our work aims to support multi-objective resource optimization (MORO). Here, we can support any user objectives (as long as we can obtain training data for them). For ease of composition, our discussion below focuses on minimizing both the stage latency (maximum latency among its instances) and the cloud cost (a weighted sum of CPU-hour and memory-hour), the two common objectives of our users.

To achieve MORO, the resource optimizer needs to make three decisions: (1) the partition count of a stage; (2) the placement plan (PP) that maps the instances of a stage to the available machines; (3) the resource plan (RP) that determines the resources (the number of cores and memory size) assigned to each instance on a given machine. Our design is guided by two principles:

- Simplicity and Efficiency. An optimal solution to MORO may require examining all possibilities of dividing the input data into m instances and arranging them to run on some of the n machines, each using one of the r possible resource configurations. In production clusters, both m and n could be 10’s of thousands, while all the RO decisions must be made well under a second. Hence, it is infeasible to run an exhaustive search for optimal solutions.

Our design principle is to break MORO into a series of simpler problems, each of which can run very fast. First, we keep the History-Based Optimizer (HBO) that uses past experiences to recommend a partition count for a stage and an initial resource plan.
for its instances. There is a merit of learning such configurations from past best-performing runs of recurring jobs (which dominate production workloads). While the recommendations may not be optimal, they serve as a good initial solution. Second, given the output of HBO, we design an Intelligent Placement Advisor (IPA) that determines the placement plan (PP), mapping the instances to machines, by predicting latencies of individual instances. Third, our Resource Assignment Advisor (RAA) will fine-tune the resource plan (RP) for each instance, after it is assigned to a specific machine, to achieve the best tradeoff between stage latency and cost. We give a theoretical justification of our approach in Section 5.

**Fine-grained Modeling and Hierarchical MOO.** As discussed earlier, instance-level recommendations for PP and RP are key to minimizing latency and cost. To do so, we design a fine-grained model that predicts the latency of each instance based on the workload characteristics, hardware, machine states, and resource plan in use. The model will be used in IPA to develop the PP that minimizes the maximum instance latency (while using the same RP for all instances). The instance-level model will be further used in RAA to improve the RP by solving a hierarchical MOO problem: We first compute the instance-level MOO solutions that minimize the latency and cost of each individual instance. We then combine the instance-level MOO solutions into stage-level MOO solutions, and recommend one of them that determines the instance-specific resources with the best tradeoff between stage latency and cost.

Fig. 3 presents the detailed architecture that extends MaxCompute with three new components (colored in purple). The trace collector collects runtime traces, including the query traces (the query plan and operator features such as cardinality), instance-level traces (input row number and data size of an instance, and the resource plan assigned to it), and machine-level traces (machine system states and hardware type). The model server featurizes the collected traces and internally learns an instance-level latency model. The internal model gets updated periodically by retraining the collected traces and internally learns an instance-level latency model. The model server runs an instance, in terms of the CPU cores and memory size.

In this section, we present how to build fine-grained instance-level models that can be used to minimize both latency and cost of each stage. To suit the complexity of big data systems, our models capture all relevant systems aspects to support the resource optimizer to make effective recommendations.

**4.1 Multi-Channel Coverage**

It is nontrivial to predict the latency of a single instance due to many factors in big data systems. Those factors include the characteristics of the (sub)query plan running in the instance, data characteristics, resources in use, current system states, and hardware properties. Each factor alone could affect the latency of an instance, and the coefficients of multiple factors can make the latency pattern more complex. Therefore, we propose the idea of multi-channel inputs (MCI) to capture all of the above factors in our model.

**Multi-channel Inputs.** For a given stage, we extract features from all available runtime traces including the query plan, resource plan, instance-level metrics, hardware profile, and system states. We design multi-channel inputs (MCI) to group the features into five channels that characterize different factors. Fig. 4 shows the overview of the MCI design for the instances of a stage.

**Channel 1: Stage-oriented features (query plan).** Channel 1 introduces the stage-oriented features that are shared among instances in a stage. It captures the operator characteristics in an operator feature matrix (see Fig. 4) and operator dependencies in a DAG structure (Fig. 5). The characteristics of each operator are featurized by three common feature types (CT1-CT3) and the customized features (CFs). CT1 identifies the operator type and is represented as a categorical variable. CT2 captures the statistics from CBO and HBO, including the cardinality, selectivity, average row size, partition count, and cost estimation. CT3 captures the IO-related properties, including the location of data (local disk or network) and strategies for shuffling. CFs are tailored for the unique properties of operators, and each feature belongs to one particular operator.

**Channels 2-5: Instance-oriented features.** Channels 2-5 characterize individual instances (Fig. 4). Channel 2, instance meta, includes an instance’s input row number and input size captured from the underlying storage system after the data partition is determined. Channel 3, resource plan, features properties of the container that runs an instance, in terms of the CPU cores and memory size. Channel 4, machine system states, records the CPU utilization, memory utilization, and IO activities of a machine to capture its running
states. Since the system states represented by count-based measurements could be infinite, we discretize the system states to reduce the computation complexity. Channel 5, hardware type, uses the machine model to distinguish a set of hardware types.

Augmented Channel 1: Additional Instance Meta (AIM). In big data systems, CBO produces the query plan without considering the characteristics of individual instances. Therefore, instances in a stage share the same query plan features in Ch1, even though their input row numbers can differ significantly. A model built on such features may have difficulty predicting different instance latencies because most of their features are identical.

To address this issue, for each instance, we seek to enrich its query plan features with instance-level characteristics. More specifically, we augment the query plan channel of an instance by adding additional instance meta (AIM) features for each operator. AIM consists of the estimated operator input/output cardinality and costs of an individual instance. It is derived using the stage-level selectivity (Ch1), the instance meta (Ch2), and the cost model in CBO.

Take job1.stage3 in Fig. 4 for example. We first get the precise input cardinalities for operators 9 and 10 in the instance by leveraging the instance meta (Ch 2) in the corresponding data partitions. Accordingly, we calculate the output cardinality of 9 and 10 by multiplying the input cardinality and the operator selectivity. Then we derive the instance-level cardinality for the remaining operators based on the operator dependency. Finally, we recompute the operator costs in an instance by reusing CBO’s cost model. Specifically, we substitute the stage-level cardinality with the instance-level cardinality, set the partition count to one, and call the CBO’s cost model to derive the operator costs for an instance.

Note that the above approach assumes that instances in a stage share the same selectivities. Although it may not always be true, our evaluation results show that the model built under this assumption could approximate the best performance using the (unrealistic) ground-truth instance-level selectivities.

Finally, recent ML models of DBMSs [24, 37] also learn data characteristics by encoding/embedding tables. In a production system, however, many tables are not reachable due to access control for security reasons, and the overhead of representing a distributed table is much higher than in a single machine. Therefore, we assume the data characteristics are already digested by CBO in our work.

4.2 MCI-based Models

Now we propose our MCI-based models to learn instance latency.

MCI-based Modeling Framework. In the output of MCI, features in the query plan channel are represented as a DAG (Fig. 5), while the instance-oriented features are in the tabular form. To incorporate these heterogeneous data structures in model building, we design two model components, as shown in Fig. 5. The plan embedder constructs a plan embedding of the plan features in the form of an arbitrary DAG structure. It first encodes the operators into a uniform feature space by padding zeros for the unmatched customized features. To embed the query plan, it then applies a Graph Transformer Networks (GTN) [56] due to its ability to learn the DAG context in heterogeneous graphs with different types of nodes. The latency predictor is a downstream model that concatenates the plan embedding with other instance-oriented features (channels 2-5) into a big vector, and feeds the vector to a Multilayer Perceptron (MLP) to predict the instance-level latency.

Modeling Tools in Our Framework. Besides GTN, our modeling framework can accommodate other models designed for DBMSs, with necessary extensions to our graph-based MCIs. Thus, we can leverage different models and examine their pros and cons in our system. Due to space limitations, our extensions of QPPNet [24] and TLSTM [35] are deferred to Appendix C.

5 STAGE-LEVEL OPTIMIZATION

In this section, we present our Stage-level Optimizer (SO) that minimizes both stage latency and cost based on instance-level models.

During execution, once a stage is handed to the scheduler, two decisions are made: the placement plan (PP) that maps instances to machines, and the resource plan (RP) that determines the CPU and memory resources of each instance on its assigned machine. The current Fuxi scheduler [61] decides a PP for each stage (i,j) to (m1, m2, m3) as follows:

1. Identify the key resource (bottleneck) in the current cluster, e.g., CPU or IO.
2. Pick m machines with top-m lowest resource watermarks.
3. Assign instances, in order of their instance id, to the m machines, and use the same resource plan for each instance as suggested by HBO. However, its negligence of latency variance among instances leads to suboptimal decisions for PP and RP.

Example 2. Figure 6 shows how Fuxi gets a suboptimal placement plan in a toy example of sending a stage of 2 instances (i1, i2) to a cluster of 3 available machines (m1, m2, m3). Assume that the
instance latency is proportional to its input row number on the same machine, and the current key resource in the cluster is the CPU. In this example, Fuxi first picks $m_1$ and $m_2$ as machines with the top-2 lowest CPU utilization (watermarks) and assigns $i_1$ to $m_3$ and $i_2$ to $m_2$ respectively. The stage latency, i.e., the maximum instance latency, is 24s. However, the optimal placement plan could achieve a 16s stage latency by assigning $i_1$ to $m_3$ and $i_2$ to $m_1$. Further, using the same resources for $i_1$ and $i_2$ is not ideal. Instead, an optimal resource plan would be adding resources to $i_2$ and reducing resources for $i_1$ so as to reduce both latency and cost, indicating the need for instance-resource specific allocation.

5.1 MOO Problem and Our Approach

To derive the optimal placement and resource plans, we begin by providing the mathematical definition of multi-objective optimization (MOO) and present an overview of our approach.

Instance-level MOO. First, consider a given instance to be run on a specific machine. We use $f_1, \ldots, f_k$ to denote the set of predictive models of the $k$ objectives and $\theta$ to denote a resource configuration available on that machine. Then the instance-level multi-objective optimization (MOO) problem is defined as:

Definition 5.1. Multi-Objective Optimization (MOO).

$\arg \min_{\theta} \begin{bmatrix} f_1(\theta) \\ \vdots \\ f_k(\theta) \end{bmatrix}$

$s.t. \quad \theta \in \Sigma \subseteq \mathbb{R}^d$

where $\Sigma$ denotes the set of all possible resource configurations.

A point $f' \in \mathbb{R}^k$ Pareto-dominates another point $f''$ iff $\forall i \in [1,k], f'_i \leq f''_i$ and $\exists j \in [1,k], f'_j < f''_j$. A point $f'$ is Pareto Optimal if there does not exist another point $f''$ that Pareto-dominates it. Then, the Pareto Set $F = \{f^1, f^2, \ldots\}$ includes all the Pareto optimal points in the objective space $\Phi \subseteq \mathbb{R}^k$, under the configurations $[\theta^1, \theta^2, \ldots]$, and is the solution to the MOO problem.

Stage-level MOO. We next consider the stage-level MOO problem over multiple instances. Consider $m$ instances, $(x_1, \ldots, x_m)$, and $n$ machines, $(y_1, \ldots, y_n)$. We use $x_i$ to denote the characteristics of $x_i$ based on its features of Ch1 and Ch2 in its multi-channel representation, and $y_j$ to denote the features of Ch4 and Ch5 of machine $y_j$. Then consider two sets of variables, $B$ and $\Theta$:

$a) \quad B \in \mathbb{R}^{m \times n}$ is the binary assignment matrix, where $B_{i,j} = 1$ when $x_i$ is assigned to $y_j$, and $\sum B_{i,j} = 1, \forall i = 1 \ldots m$.

$b) \quad \Theta = [\Theta_1, \Theta_2, \ldots, \Theta_m], \text{denotes the collection of resource configurations of } m \text{ instances, where } \forall i \in [1, \ldots, m], \Theta_i \in \Sigma_i \subseteq \mathbb{R}^d$, and $\Sigma_i$ is set of possible configurations of $d$ resources (e.g., $d = 2$ for CPU and memory resources) for instance $i$.

Given these variables, suppose that $f$ is the instance-level latency prediction model, i.e., $f(x_i, \Theta_i, y_j)$ gives the latency when $x_i$ is running on $y_j$ using the resource configuration $\Theta_i$. Then the stage-level latency can be written as, $L_{stage} = \max_{i,j} B_{i,j} f(x_i, \Theta_i, y_j)$. The stage cost is the weighted sum of cpu-hour and memory-hour and can be written as, $C_{stage} = \sum_{i,j} B_{i,j} f(x_i, \Theta_i, y_j)(w \cdot \Theta_i^T)$, where $w$ is the weight vector over $d$ resources and $w \cdot \Theta_i^T$ is the dot product between $w$ and $\Theta_i$. Other objectives can be written in a similar fashion. Then we have the Stage-level MOO Problem:

Definition 5.2. Stage-Level MOO Problem.

$\arg \min_{B, \Theta} \begin{bmatrix} L(B, \Theta) = \max_{i,j} B_{i,j} f(x_i, \Theta_i, y_j) \\ C(B, \Theta) = \sum_{i,j} B_{i,j} f(x_i, \Theta_i, y_j)(w \cdot \Theta_i^T) \end{bmatrix}$

$s.t. \quad B_{i,j} \in \{0, 1\}, \forall i = 1 \ldots m, \forall j = 1 \ldots n$

$\sum_{j} B_{i,j} \Theta_i^j \leq U_i^j, \ldots, \sum_{i} B_{i,j} \Theta_i^j \leq U_j^d, \forall j = 1 \ldots n$

where $U_j \in \mathbb{R}^d$ is the $d$-dim resource capacities on machine $y_j$.

Existing MOO Approaches. Given the above definition of stage-level MOO, one approach is to call existing MOO methods [7, 25, 28, 34] to solve it directly. However, this approach is facing a host of issues: (1) The parameter space is too large. In our problem setting both $m$ and $n$ can reach $10^5$s of thousands. Hence, both $B \in \mathbb{R}^{m \times n}$ and $\Theta = [\Theta_1, \Theta_2, \ldots, \Theta_m], \forall \Theta_i \in \mathbb{R}^d$, involve $O(mn)$ and $O(md)$ variables, respectively, which challenge all MOO methods. (2) There are also constraints specified in Def. 5.2. Most MOO methods do not handle such complex constraints and hence may often fail to return feasible solutions. We will demonstrate the performance issues of this approach in our experimental study.

Our MOO Approach. To solve the complex stage-level MOO problem while meeting stringent time constraints, we devise a novel MOO approach that proceeds in two steps:

Step 1 (IPA): Take the resource configuration $\Theta_i$ returned from the HBO optimizer as the default and assign it uniformly to all machines, $\Theta_i = \Theta_0, \forall i \in [1, \ldots, m]$. Then minimize over $B$ in Def. 5.2 by treating $\Theta_0$ as a constant.
Step 2 (RAA): Given the solution from step 1, we now minimize the variables in Def. 5.2 by treating $B^*$ as a constant.

The intuition behind our approach is that if we start with a decent choice of $\Theta_0$, as returned by HBO, we hope that step 1 will reduce stage latency via a good assignment of instances to machines by considering machine capacities and instance latencies. Then step 2 will fine-tune the resources assigned to each instance on a specific machine to reduce stage latency, cost, as well as other objectives. We next present our IPA and RAA methods that implement the above steps, respectively, prove their optimality based on their respective definitions, and optimize them to run well under a second.

5.2 Intelligent Placement Advisor (IPA)

Problem Setup. IPA determines the placement plan (PP) by way of minimizing the stage latency. We assume that the total available resources in a cluster are more than a stage’s need – this assumption is likely to be met in a production system with 10's of thousands of machines\(^1\). We further suppose that all the instances of a stage can start running simultaneously and hence the stage latency equals the maximum instance latency.

IPA begins by feeding the MCI features of the stage to the latency model and builds a latency matrix, $L \in \mathbb{R}^{m \times n}$, to include the predictions of running the instances on all available machines, i.e., $L_{i,j}$ is the latency of running instance $x_i$ on machine $y_j$. By setting $L_{i,j} = f(x_i, \Theta_0, y_j)$, ignoring the constant $\Theta_0$, and focusing on the stage latency objective in Def. 5.2, we obtain:

$$\arg\min_B \left[ \sum_{i,j} L_{i,j} B_{i,j} \right]$$

subject to:

$$\sum_i B_{i,j} = 1, \forall i \quad \text{and} \quad \sum_j B_{i,j} \leq \beta_j, \forall j$$

Here, we add the constraint $\sum_i B_{i,j} \leq \beta_j$ to meet the resource capacity constraint of each machine and a diverse placement preference (soft constraint) of sending instances to different machines to reduce resource contention. Let $\beta_j$ denote the maximum number of instances a machine can take based on its capacity constraint ($U_j \in \mathbb{R}^d$) and diversity preference. We have

$$\beta_j = \min \{U_{1,j} / \Theta_0^1, U_{2,j} / \Theta_0^2, ..., U_{d,j} / \Theta_0^d, \alpha\}$$

where $\alpha$ is a parameter that defines the maximum number of instances each machine can take based on the diverse placement preference. Basically, a smaller $\alpha$ shows a stronger preference of the diverse placement for a stage and we have $\alpha = [m/n]$.

From Eq. (3), given that the variable $B$ is a binary matrix, it is an NP-hard Integer Linear Programming (ILP) problem [4].

IPA Method. We design a new solution to IPA based on the following intuition: Since the stage latency is the maximum instance latency, intuitively, the placement plan wants to reduce the latency of the longer-running instances by sending them to the machines where they can run faster, potentially at the cost of compromising the latency of other short-running instances.

Our IPA method works as follows: We first prioritize the instances by their best possible latency (BPL), where BPL is defined as the minimum latency that an instance can achieve among all available machines. Then we keep sending the instance of the largest BPL to its matched machine and updating the BPL for instances when a machine cannot take more instances. The full procedure is given in Algorithm 1. The time complexity is $O(m(n + m + d))$ using parallelism and vectorized computations in the implementation.

**Optimality Result.** Our proof of the IPA result is based on the following column-order assumption about the latency matrix $L$: For a given column (machine), when we visit the cells in increasing order of latency, let $s_1, \ldots, s_m$ denote the indexes of the cells retrieved this way. Then the column-order assumption is that all columns of the $L$ matrix share the same order, $s_1, \ldots, s_m$. See Fig. 6 for an example of $L$ matrix. This assumption is likely to hold because, in the IPA step, all machines use the same amount of resources, $\Theta_0$, to process each instance. Hence, the order of latency across different instances is strongly correlated with the size of input tuples (or tuples that pass the filter) in those instances, which is independent of the machine used. Empirically, we verified that this assumption holds over 88-96% stages across three large production workloads, where the violations largely arise from the uncertainty of the learned model used to generate the $L$ matrix.

Below is our main optimality result of IPA. All proofs in this paper are left to Appendix D due to space limitations.

**Theorem 5.1.** IPA achieves the single-objective stage-latency optimality under the column-order assumption.

**Boosting IPA with clustering.** A remaining issue is that a stage can be up to 80K instances in a production workload, and the number of machines can also be tens of thousands. To further reduce the time cost, we exploit the idea that groups of machines or instances may behave similarly in the placement problem. Therefore, we boost the IPA efficiency by clustering both machines and instances without losing much stage latency performance.

While there exist many clustering methods [1], running them on many instances with large MCI features is still an expensive operation for a scheduler. This motivates us to design a customized clustering method based on MCI properties. An instance in a stage is characterized by its query plan (Ch1), instance meta (Ch2), resource plan (Ch3) and additional instance meta (AIM), where Ch1 and Ch3 are the same for all instances, hence not needed in clustering, and AIM fully depends on Ch1 and Ch2. So the key factors lie in Ch2, where the input row number and input size are correlated. Thus,

\(^1\)Otherwise, there is a scheduling delay for the stage with the admission control.
we approximately characterize an instance only by its input row number and apply 1D density-based clustering [1]. To represent a cluster, we choose the instance with the largest input row number to avoid latency underestimation. A machine in the cluster is characterized by its system states (Ch4) and hardware type (Ch5). We cluster them based on discretized values of Ch4 and Ch5.

Suppose that clustering yields $m'$ instance clusters and $n'$ machine clusters. Then the time complexity of IPA is $O(m \log m + n \log n + m'(n' + n') + d)$, where a sorting-based method is used for clustering both instances and machines. Since $m' \ll m, n' \ll n$, the complexity reduces to $O(m \log m + n \log n)$. Compared to other packing algorithms [16] that solve linear programming problems with quadratic complexity, the complexity of our algorithm is lower.

### 5.3 Resource Assignment Advisor (RAA)

After IPA determines the placement plan of a stage, each of its instances is scheduled to run on a specific machine. Then we propose a Resource Assignment Advisor (RAA) to tune the resource plan of each instance to solve a MOO problem, e.g., minimizing the stage latency, cloud cost, as well as other objectives.

**Stage-level MOO.** Consider the stage-level MOO problem defined in Def. 5.2. By ignoring the constant $B$, we can rewrite it in the following abstract form using aggregators $(g_1, ..., g_k)$, each for one objective, to be applied to $m$ instances:

$$\arg\min_{\Theta} F^\Theta = F(\Theta) = \left[\begin{array}{c} F_1(\Theta) = g_1(f_1(\Theta_1), ..., f_1(\Theta_m)) \\ \cdots \\ F_k(\Theta) = g_k(f_k(\Theta_1), ..., f_k(\Theta_m)) \end{array}\right]$$

where $\Theta = \{\Theta_1, \Theta_2, ..., \Theta_m\}$ is the stage-level resource configuration as $\Theta = \{\theta_1^1, \theta_1^2, ..., \theta_1^m\}$, with its corresponding $F^\Theta$ in the objective space $F \subseteq \mathbb{R}^k$. Then we can define stage-level Pareto Optimality and the Pareto Set similarly as before.

Note that we are particularly interested in two aggregators. The first is max: the stage-level value of one objective is the maximum of instance-level objective values over all instances, e.g., latency. The second is sum: the stage-level value of one objective is the sum of instance-level objective values, e.g., cost.

Example. In Figure 7, we have $f_1$ as the predictive model for latency ($g_1 = \text{max}$) and $f_2$ for cost ($g_2 = \text{sum}$). Suppose $\Theta = \{\theta_1^1, \theta_1^2\}$ as the stage-level resource configuration. Then we have the stage-level latency as $F_1(\Theta) = \text{max}(150, 100) = 150$, the stage-level cost as $F_2(\Theta) = \text{sum}(5, 5) = 10$, and the stage-level solution $F^\Theta$ is $[150, 10]$.

While one may consider using existing MOO methods to solve Eq. (4), it is still subject to a large number ($O(md)$) of variables. To enable a fast algorithm, we next introduce our Hierarchical MOO approach developed in the divide-and-conquer paradigm.

**Definition 5.3. Hierarchical MOO.** We solve the instance-level MOO problem for each of the $m$ instances of a stage separately, for which we can use any existing MOO method (in practice, the Progressive Frontier PF algorithm [34] since it is shown to be the fastest). Suppose that there are $k$ stage-level objectives, each with its max or sum aggregator to be applied to $m$ instances. Our goal is to efficiently find the stage-level MOO solutions from the instance-level MOO solutions, for which we use $f_i^j$ to denote the $j$-th Pareto-optimal solution in the $i$-th instance by using $\theta_i^j$.

**General hierarchical MOO solution.** Suppose that there are $k$ user objectives, where $k_1$ objectives use the max and $k_2$ objectives use sum, $k_1 + k_2 = k$. We are also given instance-level Pareto sets. Then we want to select a solution for each instance such that the corresponding stage-level solution is Pareto optimal. Algorithm 2 gives the full description. After initialization, line 2 obtains the lower and upper bounds for each of the $k_1$ max objectives. In line 3 ($\text{find\_all\_possible\_values}$), we first find for each max objective all the possible values as one list, and then use the Cartesian product of these lists as the candidates for the $k_1$ max objectives. In lines 4 to 10, given one candidate, we try to find the corresponding Pareto-optimal solution for the $k_2$ sum objectives. For the $k_2$ objectives using sum, due to the complexity of the sum operation, we cannot afford the enumeration, which grows exponentially in the number of instances, $O(p_{max})$ where $p_{max}$ is the maximum number of instance-level Pareto points among $m$ instances. To reduce the complexity, we resort to any existing MOO method, denoted by the function $\text{find\_optimal}$, that (in line 6) for each instance selects one Pareto-optimal solution for the $k_2$ objectives. At the end of the procedure, we add a filter to remove the non-optimal solutions.

Example. In Figure 7, we calculate the lower and upper bounds of the stage-level latency as max(55, 100) = 100 and max(150, 300) = 300 respectively. Then we enumerate all the possible stage-level latency values within the bounds (100, 150, 300) to collect potential Pareto-optimal solutions. For latency=100, there is only one feasible choice $\Theta = \{\theta_1^1, \theta_1^2\}$ and hence the only solution is [100, 25]. Similarly for latency=150, we get another solution [150, 10]. When latency=300, there are two solutions: [300, 24] with $\Theta = \{\theta_1^1, \theta_1^2\}$ and [300, 9] with $\Theta = \{\theta_1^1, \theta_1^2\}$. The first one will be filtered because it is dominated by the latter one (line 6). Finally, we further filter solutions being dominated in the chosen set (line 11) and get the stage-level MOO solutions as [[100, 25], [150, 10], [300, 9]].

**Proposition 5.1.** For a stage-level MOO problem, Algorithm 2 guarantees to find a subset of stage-level Pareto optimal points.
Figure 8 shows an example of the Pareto sets of Multi-channel Inputs Performance Profiling. We start the procedure by selecting the first Pareto solution whose objectives could achieve the smallest distance to the Utopia point, which is the hypothetical optimal in all objectives. For each instance of the stage, it recommends the resource plan Pareto set, we reuse UDAO's Weighted Utopia Nearest (WUN) strategy with Clustering. Proposition 5.2. It terminates until there is no entry below the current \( \theta \) as \( \beta \) is corresponding to \( \beta \) that is below it in the same Pareto set. Now we have the second solution \( \beta \). As \( \beta \) is the number of instance-level Pareto solutions \( \beta \), \( \beta \) is a state, where \( \beta \) is a step such that the \( \beta \)th dimension in state \( \beta \) is increased by 1. (4) \( \beta \) is the number of instance-level Pareto solutions for instance \( \beta \). Then Algorithm 3 gives the RAA Path algorithm.

Proposition 5.2. For a stage-level MOO problem with two objectives using max and sum, respectively, Algorithm 3 guarantees to find the full set of stage-level Pareto optimal points at the complexity of \( O(m \cdot \beta \max \log (m \cdot \beta \max)) \).

Algorithm 3: RAA Path policy to construct \( PO_\beta \) and \( PO_F \)

Require: \( p_i, \beta_i, f_i', V_i \in [1, m], j \in [1, p_i] \), where we pre-sort \( f_i' \) and \( \beta_i \) in the descending order of latency for each instance.

1. \( u_i' = \text{lat}(f_i'), V_i \in [1, m], j \in [1, p_i] \).
2. \( PO_\beta = \{ \}, PO_F = \{ \}, \lambda = [1, 1, \ldots, 1], \beta \max = \infty \).
3. Build the max heap \( W \) with data points \( \{u_i', \beta_i\} \), \( \forall V_i \in [1, m] \).

4. repeat
   5. \( qmax, \lambda = \text{Q.pop()} \)
   6. if \( qmax < \beta \max \) then
      7. \( PO_\beta.\text{append}(\Theta^\lambda), PO_F.\text{append}(F(\Theta^\lambda)), \beta \max = qmax \)
   8. end if
   9. \( \lambda = \pi_1(\lambda) \)
   10. if \( \lambda_i > p_i \) then return \( PO_\beta, PO_F \)
   11. end if
   12. \( Q.\text{push}((u_i', \beta_i)) \)
5. until True

6.1 Model Evaluation

To train models, we partition the traces of workloads A-C into training, validation, and test sets, and tune the hyperparameters of all models using the validation set. As the default, we train MCI-GTNN over all channels augmented by AIM for the instance latency prediction. More details about training are given in Appendix F.3.

We report model accuracy on 5 metrics: (1) weighted mean absolute percentage error (WMAPE), (2) median error (MdErr), (3) 95th percentile Error (95%Err), (4) Pearson correlation (Corr), and (5) error of the cloud cost (GlbErr). We choose WMAPE as the primary, also the hardest, metric because it assigns more weights to long-running instances, which are more important in resource optimization.

Expt 1: Performance Profiling. We report the performance of our best models for each workload in Table 3. First, our best model achieves 9-19% WMAPEs and 7-15% MdErrs over the three workloads with our MCI features (Ch1-Ch5 plus AIM). Second, WMAPE is a more challenging metric than MdErr. The distribution of the instance latency in production workloads is highly skewed to short- and median-running instances. Therefore, MdErr is determined by the majority of the instances but does not capture well those long-running ones. Third, the error rate of the total cloud cost is 3.4-5.5 smaller than WMAPEs, because individual errors of single instances could have a canceling effect on the global resource metric, hence better model performance on the global metric.

Our breakdown analyses further show that IO-intensive operators and the dynamics of system states are the two main sources of model errors. First, by training a separate model for the instance-level operator latency and calculating the error contribution of each operator type, we find the top-3 most inaccurate operators as StreamLineWrite, TableScan, and MergeJoin, all involving frequent IO activities. It is likely that current traces in MaxCompute miss the features to fully characterize IO operations. As for the system dynamics, we train a separate model by augmenting system states with the average system metrics during the lifetime of an instance (which is not realistic to get before running an instance). But this allows us to show that the model further reduces WMAPE and MdErr to 6-12% and 5-10%, pointing the error source to the system dynamics. The comprehensive results are shown in Appendix F.6.

Expt 2: Multi-channel Inputs. We now investigate the importance of each channel to model performance. We train separate models

|       | Num Jobs | Num Stages | Num Inst | Stages/job | Stages/inst | Ops/stage | Avg Job Lat(s) | Avg Stage Lat(s) | Avg Inst Lat(s) |
|-------|----------|------------|----------|------------|------------|-----------|----------------|-----------------|-----------------|
| A     | 405K     | 970K       | 34M      | 2.40       | 35.43      | 3.71      | 30.97          | 14.64           | 16.85           |
| B     | 173K     | 858K       | 36M      | 4.95       | 42.02      | 6.27      | 120.15         | 39.72           | 15.63           |
| C     | 41K      | 100K       | 50M      | 2.42       | 565.51     | 5.31      | 376.83         | 181.88          | 71.08           |

Table 1: Workload statistics for 3 workload over 5 days

6 EXPERIMENTS

This section presents the evaluation of our models and stage optimizer. Using production traces from MaxCompute, we first analyze our models and compare them to state-of-the-art modeling techniques. We then report the end-to-end performance of our stage optimizer using a simulator of the extended MaxCompute environment (detailed in F.2) by replaying the production traces.

Workload Characteristics. See Table 1 for descriptions of production workloads A-C. We give additional details in Appendix F.1.
Training Overhead and Model Adaptivity

Comparison with QPPNet

Impact of Cardinality.

IPA Only.

WMAPE

10%

20%

30%

40%

5%

10%

8.5%

9.0%

14.2%

24.3%

22.5%

19.1%

all_on+calib

all_on

ch5_off

ch3_off

ch4_off

ch1_off

ch2_off

Expt 5:

Fig. 9(a) shows our results. The top-3 important features are the instance meta (Ch2), the query plan (Ch1), and the system states (Ch4). By turning off each, WMAPE gets worse by 18-66%, 16-50%, and 9-27%, respectively, compared to all_on. The effect of the hardware type (Ch5) is not significant, likely because the hardware types used are all high-performance ones. The effect of the resource plan (Ch3) is small here due to its sparsity in the feature space; e.g., only 26 different resource plans are observed in workload B. But their effects will be different once they are tuned widely in MOO.

Expt 3: Impact of Cardinality. We train separate models by deriving AIMs from different cardinalities: (1) all_on+calib represents the stage-oriented cardinality from MaxCompute’s CBO; (2) all_on+simu1 represents the ground-truth stage-level cardinalities while assuming operators in different instances share the stage selectivities; (3) all_on+simu2 applies the ground-truth instance-level cardinalities for operators. Fig. 9(b) shows at most a 0.2% and 0.4% WMAPE can be reduced by using all_on+simu1 and all_on+simu2, respectively. This means that improving cardinality estimation alone cannot improve much latency prediction in big data systems, which is consistent with CLEO’s observation [33].

Expt 4: Comparison with QPPNet [24] and TLSTM [35]. We next compare different modeling tools, including (1) original QPPNet, (2) original TLSTM, (3) our extension, MCI-based QPPNet, (4) MCI-based TLSTM, and (5) MCI+GTN (our new graph encoder). Fig. 9(c) shows our results. First, QPPNet and TLSTM achieve 22-36% and 15-31% WMAPE, respectively, 2-3x larger than our best model, MCI+GTN. Second, using our MCI, MCI+QPPNet and MCI+TLSTM can improve WMAPE by 12-15% and 6-12%, respectively, while MCI+TLSTM and MCI+GTN achieve close performance. Thus, our MCI framework offers both good modeling performance and extensibility to adapt SOTAs from a single machine to big data systems.

Expt 5: Training Overhead and Model Adaptivity. We next report on the training overhead. The data preparation stage costs ~2 minutes every day to collect new traces from each department using MaxCompute jobs (with a parallelism of 219). We collected 5-day traces from three internal departments, totaling 0.62M jobs, 2M stages, and 407G bytes. In the training stage, we run 24h periodic retraining (retrain) at midnight of each day when the workloads are light, which takes 3.5 hours on average over 16 GPUs (including hyperparameter tuning). Optionally, we run fine-tuning every 6 hours (retrain+finetune), at the average cost of 0.9h each.

To study model adaptivity, we design two settings of workload drifts: (a) a realistic setting where queries from 5 days are injected in temporal order; (b) a hypothetical worst case where queries are injected in decreasing order of latency. In both settings, we compare a static method that trains the model using the data from the first 6 hours and never updates it afterward, with our retrain and retrain+finetune methods. Our results, as shown in Fig. 10 for workload C, include: (1) The static approach can reach up to 72% WMAPE in some hours of a day, demonstrating the presence of workload drifts in our dataset. (2) retrain and retrain+finetune adapt better to the workload drifts, keeping errors mostly in the range of 15-25% after days of training, while having tradeoffs between them: If the workload patterns in 24h windows are highly regular, retrain works slightly better than retrain+finetune by avoiding overfitting to insignificant local changes (workload A). Otherwise, retrain+finetune works better by adapting to significant local changes (workload B). More discussion is given in Appendix F.4.3.

Additional experiments are in Appendix F.7 and F.8.

6.2 Resource Optimization (RO) Evaluation

We next evaluate the end-to-end performance of our Stage Optimizer (SO) against the current HBO and Fuxi scheduler [61], as well as other MOO methods using production workloads A-C. As we cannot run experiments directly in the production clusters, we developed a simulator of the extended MaxCompute and replayed the query traces to conduct our experiments (detailed in F.2).

We consider the following metrics in resource optimization (RO): (1) coverage, the ratio of stages that receive feasible solutions within 60s; (2) Lat\(_{\text{in}}^{i}\), the average stage latency that includes the RO time; (3) Cost\(_{\text{avg}}\), the average cloud cost of all stages in a workload; (4) T\(_{\text{RO}}\), the RO time cost. Below, we first present a microbenchmark of our methods and other MOO methods using 29 subworkloads in Table 2, and then report our net benefit over the full dataset in Table 4.

Expt 6: IPA Only. We first turn on only the IPA module in the stage optimizer (no RAA) and compare the two options of IPA to the Fuxi scheduler over the 29 subworkloads. As shown in Tab. 2, the clustered version IPA(Cluster) reduces the stage latency (including the solving time) by 12-50% and cost by 4-14%, with the average time cost between 10-33 msec. Without clustering, IPA(Org)
achieves comparable reduction rates as IPA(Cluster) but costs 2-8x more time for solving.

Expt 7: IPA+RAA. We now run IPA(Cluster) with RAA of four choices. IPA+RAA(Path) solves the resource plan by applying RAA Path over instance and machine clusters and achieves the best performance. It reduces the stage latency by 36-80% and cloud cost by 29-75%, with an average time cost between 17-156ms (including DBSCAN for the instance clustering, which incurs up to 937 msec overhead (up to 19s for a stage). IPA+RAA(DBSCAN) applies DBSCAN for the instance clustering, which incurs up to 937 msec for a stage, hence inefficient for production use. IPA+RAA(W/O_C) does RAA without clustering and suffers from the distribution within $\mu$ of the actual latency, and samples from the distribution within $\mu \pm 3\sigma$. Table 4 shows that in both noisy and noise-free cases. For the noisy case, we turn on the extra 2D MOO prob-

Table 2: Average Reduction Rate (RR) against Fuxi in 29 subworkloads within 60s

| SO choice          | Coverage | Lat\textsuperscript{(m)} | Costs\textsubscript{\textsuperscript{avg}} | any(T\textsubscript{s}) / | max(T\textsubscript{s}) / |
|--------------------|----------|--------------------------|-----------------------------|-------------------------|-------------------------|
| A                  | B        | C                        | A                           | B                       | C                       |
| IPA(Org)           | 100%     | 100%                     | 100%                        | 100%                    | 100%                    |
| IPA(Cluster)       | 100%     | 100%                     | 100%                        | 100%                    | 100%                    |
| IPA+RAA(W/O_C)     | 100%     | 100%                     | 8%                          | 79%                     | 58%                     |
| IPA+RAA(DBSCAN)    | 100%     | 100%                     | 27%                         | 69%                     | 67%                     |
| IPA+RAA(Generic)   | 100%     | 100%                     | 36%                         | 80%                     | 76%                     |
| IPA+RAA(Path)      | 100%     | 100%                     | 100%                        | 100%                    | 100%                    |

Table 3: Modeling Performance

| Bootstrap Model    | Stage Lat (ms): | Cost: |
|--------------------|-----------------|-------|
| GTN+MCI            | 34%             | 49%   |
| TLSTM              | 17%             | 24%   |
| QPPNet             | 34%             | 15%   |

Table 4: Average RR over 2M stages

7 CONCLUSIONS

Table 1

| Bootstrap Model    | Stage Lat (ms): | Cost: |
|--------------------|-----------------|-------|
| IPA(noise-free)    | 10%, 15%, 44%   | 3%    |
| IPA(noisy)         | 9%, 10%, 42%    | 3%    |


Expt 8: MOO baselines. We next compare to SOTA MOO solutions, EVO [7], WS(Sample) [25], and PF (MOOGD) [34], using Def. 5.2. See Appendix A for their implementation details. As shown in the 3 red rows of Table 2, (1) over 29 sub-workloads, none of them guarantees to return all results within 60s; (2) their latency and cost reduction rates are all dominated by IPA+RAA(Path), and even lose to the Fuxi scheduler on some workloads; (3) their solving time is 1-2 magnitude higher than our approach, making it infeasible to be used by a cloud scheduler. As an alternative, we apply IPA to solve the $B$ variables and these MOO methods to solve only $\Theta$ based on Eq. (4). As shown in the last 3 blue rows in Table 2, they are still inferior to IPA+RAA(Path) in both latency and cost reduction and in running time (mostly taking 1-6 sec to complete).

Expt 9: Net Benefits. We next compare our SO (IPA+RAA) against Fuxi’s scheduling results by considering the model effects: the noise-free case means that the predicted latency is the true latency, while the noisy case captures the fact that the true latency is different from the predicted one. We run the entire 2M stages over 3 departments in both noisy and noise-free cases. For the noisy case, we turn on the actual latency simulator (GPR model) to simulate the actual latency. Specifically, given a predicted instance latency, GPR generates a Gaussian distribution ($N(\mu, \sigma)$) of the actual latency, and samples from the distribution within $\pm \sigma$. Table 4 shows that in both noisy-free and noisy settings, SO (IPA+RAA) significantly reduces stage latency and cloud cost compared to Fuxi, with the fine-grained MCI+GTN model for latency prediction.

Expt 10: Impact of Model Accuracy. Finally, to quantify the impact of model accuracy on resource optimization, we use GPR models pre-trained over three bootstrap models (MCI+GTN, TLSTM, QPPNet). Note that in terms of model accuracy, we have (MCI+GTN > TLSTM > QPPNet), as shown in Fig. 9(c). We borrow Fuxi’s ground-truth placement plan and ask RAA for resource plans of each stage under the same condition of the system states as Fuxi. To be fair in the comparison, we dropped the scheduling delays for all stages and set the stage latency as the maximum instance latency. Table 4 compares the RAA’s reduction rate (RR) among different bootstrap models. These results show that indeed, better reduction rates are achieved by using a more accurate model, which validates the importance of having a fine-grained accurate prediction model.

ACKNOWLEDGMENTS

This work was partially supported by the European Research Council (ERC) Horizon 2020 research and innovation programme (grant n725561), Alibaba Group through Alibaba Innovative Research Program, and China Scholarship Council (CSC). We also thank Yongfeng Chai, Daoyuan Chen, Xiaozong Cui, Botong Huang, Xiaofeng Zhang, and Yang Zhang from the Alibaba group for the discussion and help throughout the project.
A PROBLEM FORMULATION

Consider $m$ instances ($x_1, ..., x_m$) and $n$ machines ($y_1, ..., y_n$). Denote $\hat{x}_i$ as the features of Ch1 and Ch2 of instance $x_i$, and $\hat{y}_j$ as the features of Ch4 and Ch5 of machine $y_j$. Given two sets of variables $B$ and $\Theta$, assuming that $f$ is the instance-level latency prediction model, we have $L_{ij} = f(\hat{x}_i, \Theta_i, \hat{y}_j)$ as the latency when $x_i$ is running on $y_j$ using the resource configuration $\Theta_i$.

Denote two objectives as the stage-level latency

$$L_{stage} = L(B, \Theta) = \max_{i,j} B_{ij} f(\hat{x}_i, \Theta_i, \hat{y}_j)$$

and the stage-level total cost (the weighted sum of cpu-hour and memory-hour)

$$C_{stage} = C(B, \Theta) = \sum_{i,j} B_{ij} L_{ij}(w \cdot \Theta_{ij}^T) = \sum_{i,j} B_{ij} f(\hat{x}_i, \Theta_i, \hat{y}_j)(w \cdot \Theta_{ij}^T)$$

where $w_1$ and $w_2$ are two constants. $w$ is a weight vector. Similarly, other objectives such as CPU-hour, memory-hour can be defined.

Consider $m$ instances ($x_1, ..., x_m$) and $n$ machines ($y_1, ..., y_n$). Given two set of variables $B$ and $\Theta$, $f$ is instance-level latency when $x_i$ is running on $y_j$ using the resource configuration $\Theta_i$. We have a constrained Multi-Objective Optimization (MOO) problem.

$$\arg \min_{B, \Theta} \begin{bmatrix} L(B, \Theta) = \max_{i,j} B_{ij} f(\hat{x}_i, \Theta_i, \hat{y}_j) \\ C(B, \Theta) = \sum_{i,j} B_{ij} f(\hat{x}_i, \Theta_i, \hat{y}_j)(w \cdot \Theta_{ij}^T) \end{bmatrix}$$

(5)

where

$$\begin{array}{l}
B_{ij} \in \{0,1\}, \forall i = 1..m, \forall j = 1..n \\
\sum_{j} B_{ij} \Theta_{i,j}^1 \leq U_{i,j}, \forall i = 1..m \\
\sum_{i} B_{ij} \Theta_{i,j}^2 \leq U_{j}^d, \forall j = 1..n \\
\sum_{i} B_{ij} \leq \alpha, \forall j = 1..n
\end{array}$$

where $U_j \in \mathbb{R}^d$ is the capacity of $d$ resources on machine $y_j$ and $\alpha$ defines the maximum number of instances each machine can take based on the diverse placement preference. Basically, a smaller $\alpha$ shows a stronger preference of the diverse placement for a stage and we have $\alpha = \lceil m/n \rceil$.

To simplify the problem, we take two steps:

Step 1: Take the resource configuration $\Theta_0$ returned from the HBO optimizer as the default and assign it uniformly to all instances, $\Theta_i = \Theta_0, \forall i \in \{1, ..., m\}$. Then minimize over $B$ in Eq. (5) by treating $\Theta_0$ as a constant. Then the objective functions are shown in Eq. (6).

$$\arg \min_B \begin{bmatrix} L(B, \Theta_0) = \max_{i,j} B_{ij} f(\hat{x}_i, \Theta_0, \hat{y}_j) \\ C(B, \Theta_0) = \sum_{i,j} B_{ij} f(\hat{x}_i, \Theta_0, \hat{y}_j)(w \cdot \Theta_{ij,0}^T) \end{bmatrix}$$

(6)

Step 2: Given the solution from step 1, $B^*$, minimize over $\Theta$ by treating $B^*$ as a constant, as shown in Eq. (7).

$$\arg \min_{\Theta} \begin{bmatrix} L(B^*, \Theta) = \max_{i,j} B_{ij}^* f(\hat{x}_i, \Theta_i, \hat{y}_j) \\ C(B^*, \Theta) = \sum_{i,j} B_{ij}^* f(\hat{x}_i, \Theta_i, \hat{y}_j)(w \cdot \Theta_{ij}^T) \end{bmatrix}$$

(7)

The alternative formula forms are also shown in Eq. (3) for the step 1 and Eq. (4) for the step 2. The intuition behind our approach is that if we start with a decent choice of $\Theta_0$, as returned by HBO, we hope that step 1 will reduce stage latency via a good assignment of instances to machines by considering machine capacities and instance latencies. Then step 2 will fine-tune the resources assigned to each instance on a specific machine to reduce stage latency, cost, as well as other objectives.

A.1 Formulation of Different MOO Problems

We consider solving the MOO problem in two scenarios:

1. Plan A: Optimization over $B$ and $\Theta$. Take the entire parameter space with $B$ and $\Theta$ as variables.
2. Plan B: Optimization over $\Theta$. Use the solution from step 1, and take Step 2 with $\Theta$ as the variable.

A.1.1 Plan A: Optimization over $B$ and $\Theta$.

We consider two methods implemented for the plan A. The first method solves the vanilla plan A by strictly applying the MOO definition in Eq. (5). However it suffers from the huge amount of variables and constraints, where we get $m \times (n + d)$ variables, and $m + (d + 1) \times n$ constraints. Notice both $m$ and $n$ can be several thousands, the problem could have millions of parameter and thousands of constraints.

To reduce the dimensionality, we consider the clustered plan A that solves the variables after clustering instances and machines as our IPA did. After clustering, there are $m'$ instance clusters ($X_{i,1}', ..., X_{i,m'}'$) and $n'$ machine clusters ($Y_{j,1}', ..., Y_{j,n'}'$), where $x_i'$ is the representative instance of instance cluster $X_{i,j}'$, $y_j'$ is the representative machine of machine cluster $Y_{j,j}'$. Denote $\tilde{x}_i'$ as the features of Ch1 and Ch2 of $x_i'$, and $\tilde{y}_j'$ as the features of Ch4 and Ch5 of $y_j'$. Given two sets of variables $B'$ and $\Theta'$, assuming that $f$ is the instance-level latency prediction model, we have $f(\tilde{x}_i', \Theta_i', \tilde{y}_j')$ as the latency when $x_i'$ running on $y_j'$ using the resource configuration $\Theta_i'$. We assume that instances in the same instance cluster behave similarly when they are scheduled to the machines from the same cluster. Therefore, the constrained MOO problem is defined in Eq. (8).

$$\arg \min_{B', \Theta'} \begin{bmatrix} L'(B', \Theta') = \max_{i,j} f(\tilde{x}_i', \Theta_i', \tilde{y}_j) \\ C'(B', \Theta') = \sum_{i,j} f(\tilde{x}_i', \Theta_i', \tilde{y}_j)(w \cdot \Theta_{ij}^T) \end{bmatrix}$$

(8)

where $I(B'_{i,j})$ is an indicator function of $B'_{i,j}$. $|X_{i,j}'|$ denotes the number of instances in the instance cluster $X_{i,j}'$; $|Y_{j,j}'|$ denotes the number of machines in the machine cluster $Y_{j,j}'$; $\sum_{U_r \in \mathbb{R}^d}$ is the total capacity of the $d$ resources on the machine cluster $Y_{j,j}'$; $\alpha$ defines the maximum number of instances each machine cluster can take based on the diverse placement preference and we have $\alpha = \lceil m/n \rceil$.

In clustered Plan A, the number of variables and the number of constraints are reduced to $m' \times ((d + 1) \times n')$ and $m' \times ((d + 1) \times n')$, respectively. The constraints guarantee the requested resources of instances allocated in the same machine cluster do not exceed its capacity.
Table 5: Notation for the MOO problem

| Symbol | Description |
|--------|-------------|
| \( d \) | the number of resource types in a resource configuration (RC). \( d = 2 \) in our case. |
| \( m \) | the number of instances. |
| \( n \) | the number of machines. |
| \( x_i \) | the \( i \)-th instance |
| \( \tilde{x}_i \) | the features on channel 1 and channel 2 and AIM towards \( x_i \) |
| \( y_j \) | the \( j \)-th machine |
| \( \tilde{y}_j \) | the features on channel 4 and channel 5 towards \( y_j \) |
| \( B \) | the placement matrix \( B \in \{0,1\}^{m \times n} \) |
| \( B_{i,j} \) | the \( i \)-th row, \( j \)-th column of \( B \). \( B_{i,j} = 1 \) when \( x_i \) is sent to \( y_j \). |
| \( \Theta \) | the second set of parameters, a.k.a. the resource plan matrix \( \Theta \subseteq \mathbb{R}^{m \times 2} \) |
| \( \Theta_i \) | the resource plan of the \( i \)-th instance \( x_i \). \( \Theta_i \in \mathbb{R}^2 \). In our case, \( \Theta_i = (\text{req}_\text{cores}, \text{req}_\text{mem}) \) |
| \( f \) | the instance-level latency model, mapping from channels 1-5 and AIM to latency estimation. |

A.1.2 Plan B: Optimization over \( \Theta \).

To reduce the parameter space in the original problem, we resolve the \( B \) variable by IPA in the first step, and other MOO solutions to the step 2. Given the solution from step 1, \( B' \), the MOO problem in Eq. (5) changes to the following Eq. (9).

\[
\arg \min_{\Theta} \begin{bmatrix} L(B', \Theta) = \max_{i,j} B'_{i,j}(x_i, \Theta_i, \tilde{y}_j) \\ C(B', \Theta) = \sum_{i,j} B'_{i,j}(x_i, \Theta_i, \tilde{y}_j)(w \cdot \Theta_i^T) \end{bmatrix}
\]

\( \text{s.t.} \sum_i B'_{i,j} \Theta_i^T \leq U_j^*, \ldots, \sum_i B'_{i,j} \Theta_i^{d \times 1} \leq U_j^{d \times 1} \quad \forall j = 1\ldots n \)

We also consider two methods implemented for the plan B. The first method vanilla plan B solves Eq. (9) directly, where the number of variables is \( d \times m \), and the number of constraints is \( d \times n \). Notice \( m \) and \( n \) can both be tens of thousands, the dimensionality of the problem is very high.

To further reduce the number of variables and constraints, we consider the clustered Plan B that solves the problem after clustering. After clustering, consider \( m' \) instance clusters \( (X'_1,\ldots,X'_{m'}) \) and \( n \) machines \( (y_1,\ldots,y_n) \), where \( X'_i \) is the representative instance of instance cluster \( X'_i \), \( y_i \) is the allocated machine of \( X'_i \) after \( B' \) is given. Denote \( x'_i \) as the features of Ch1 and Ch2 of the representative instance \( x'_i \) in instance cluster \( X'_i \), and \( \tilde{y}_j \) as the features of Ch4 and Ch5 of the allocated machine \( y_j \).

To align to the clustering \( B' \) matrix in Plan A, we utilize \( B'^* \) to denote the placement of instances in clustered Plan B. But the dimensions of \( B' \) matrix in Plan A and \( B'^* \) in Plan B are different. The differences occur in two aspects. Firstly, the instance clustering of Plan B and Plan A are different. In Plan A, as the elements in \( B' \) matrix are variables, the instance clustering can only consider the instance’s feature (e.g., cardinality). In Plan B, the placement plan \( B'^* \) matrix is given, so the instance clustering can consider the feature of the instance and the machine state. It divided instance clusters of Plan A into more sub-clusters, which leads to different number of instance clusters (the number of rows in \( B' \) and \( B'^* \)). Furthermore, \( B'^* \) is given in Plan B, and \( B'^*_{i,j} \) is a binary value, which indicates whether the representative instance of instance cluster \( X'_i \) is scheduled on the specific machine \( y_j \) rather than a machine cluster. Therefore, the number of columns of \( B' \) matrix in Plan A and \( B'^* \) in Plan B are different.

Similarly, the variables in Plan B are \( \Theta' \), assuming that \( f \) is the instance-level latency prediction model, we have \( L'_{i,j} = f(x'_i, \Theta'_i, \tilde{y}_j) \) as the latency when \( x'_i \) is running on \( y_j \) using the resource configuration \( \Theta'_i, \tilde{y}_j \). We assume groups of instances may behave similarly. After clustering, the constrained MOO problem is defined as in Eq. (10).
We apply NSGA-II. We applied the NSGA-II family of Evolutionary (Evo) algorithms [7] for solving MOO problems.

A.2 Implementation of Existing MOO Methods

In our experiment, we consider $d = 2$ types of resources as CPU and memory. Thus, we implemented the existing MOO methods for $d = 2$. However, our implementation can be easily extended to the cases when $d > 2$.

Method 1: Evolutionary algorithm. We apply NSGA-II [?] in the family of Evolutionary (Evo) algorithms [7] for solving MOO problems of both clustered Plan A and Plan B. The objective functions and constraints are referred to the problem formulation mentioned above. We did hyperparameter tuning to choose a set of optimal population size and the number of iterations for both Plan A (EVO) and Plan B (IPA+EVO). The NSGA-II is implemented by the library Platypus [?]. Here are the technical details:

1. **Complex functions and constraints**: When calculating the stage-level function values, we need to call the predictive model, where the variables are part of the input to get predictions. Moreover, the constraints in Plan A are quadratic, which is more complex than a linear representation.

2. **Evaluation time**: As we discussed before, the number of variables and constraints are related to the number of instances. The time cost of different stages could be various if they have a very different number of instances. Based on our observation, a stage could take over 10 minutes to return solutions, which is not acceptable under the requirement of efficiency and we have 0.1M-1M stages of running. Therefore, we limit the time cost to 60 seconds for solving a stage MOO problem.

3. **Hyperparameter tuning**: Population size and the number of iterations are two hyperparameters influencing optimization performance. To select one set of optimal hyperparameters, under the time limit of 60 seconds, we try 12 sets of hyperparameters for the workload with 103 stages which include the number of instances varies from 1 to over 6k. Finally, the hyperparameter is selected based on the higher coverage of feasible stages and a higher reduction rate on the stage-level latency and cost.

4. **Randomness**: NSGA-II is an evolutionary algorithm where the population is initialized randomly, and genetic operations are executed under a probability (e.g., crossover and mutation). To make the results reproducible, we fix the randomness of each iteration related to population generation and genetic operations.

Method 2: Weighted Sum. We applied $\text{WS(Sample)}$ [25] to reduce a MOO problem into a single-objective optimization (SOO) problem by minimizing the weighted sum of the stage latency and cost. We applied the random sampling approach to solving the SOO problem in our implementation. For each stage, we get the corresponding MOO solutions over a set of weights over latency and cost by the following steps:

1. Sample up to 100K variable choices randomly.
2. Filter the variables that violate the constraints of resource capacity or the diverse preference.
3. Evaluate the objective values over each filtered variable.
4. Iterate the objective weights over latency and cost and return the variables that minimize the weighted sum of the objectives respectively.

5. Form the MOO solutions based on the returned variables. $\text{WS(Sample)}$ treats the resource plan $\Theta$ and the placement plan $\Theta_{\text{IPA}}$ as the variables in plan $A$. In plan $B$, $\text{IPA+WS(Sample)}$ solves only $\Theta$ with the placement plan returned by our IPA(Cluster).

Method 3: Progressive Frontier (PF). We implemented our previous work $\text{PF(MOGD)}$ [34] in the parallel version (PF-AP) to solve the MOO problem progressively. $\text{PF(MOGD)}$ transforms the MOO problem into a set of single-objective constraint optimization (CO) problems and solves them via the Multi-Objective Gradient Descent (MOGD) solver. When doing gradient descent for solving the variables, the MOGD solver treats $\Theta$ and $B$ as float matrices and rounds their values to integers in their corresponding domain spaces after every backward step.

B WORKLOAD DESCRIPTION

Example 1. Figure 2(a) and 2(b) show that in a production trace of 0.62 million jobs, there are 1.9 million stages in total, with up to 64 stages in each job, and 121 million instances, with up to 81430 instances in a stage. For a particular stage with 6716 instances, Figure 2(c) shows that the latencies of different instances vary a lot.
Most notably, choosing the latency of a single instance as our fine-modeling target distinguishes our work from other existing work that captures end-to-end query latency [34, 58] or operator latency including multiple instances on different machines [33]. These coarse-grained models not only are subject to highly-variable predictions but also miss opportunities for instance-level recommendations – our fine-grained models aim to solve both problems.

In addition, existing works considered only part of the system issues in resource management. For example, the scheduling algorithms [11, 61?] treat scheduling tasks as blackboxes without considering the complexity of the workloads. Recent works [2, 33, 47] optimize the resource plan by leveraging the query characteristics, but without taking into account the cluster complexity. In contrast, our work aims to capture all relevant systems issues in modeling so that the resource optimizer can leverage the model to make effective recommendations.

**Modeling Tools in Our Framework.** Besides our GTN based model, we next show that our modeling framework can accommodate other modeling tools designed for DBMSs, with necessary extensions. As thus, we can leverage different modeling tools in our framework and examine their pros and cons in experimental evaluation.

QPPNet [24] and TLSTM [35] are two state-of-the-art methods that learn the query performance in a DBMS. QPPNet learns operator-level neural networks named neural units for each query operator in a plan-structured model. The output of each neural unit includes a data vector and a latency channel. The input of each neural unit consists of the operator-oriented features and data vectors generated by its child nodes (if any). QPPNet returns the query latency prediction from the latency channel in the output of the root neural unit. TLSTM encodes query operators into a uniform feature space and feeds the operator encodings to a Tree-Structured Long Short-Term Memory Network (TreeLSTM) [37] to learn the query latency in a query plan tree. Figure 13 shows how TLSTM does stage embedding over a query job by taking use of its tree structure.

In our work, we extend QPPNet and TLSTM to support the MCI-based modeling in big data systems by solving two issues:

1. **How to model arbitrary DAG structures in a tree-based model?** Both QPPNet and TLSTM consider only tree-structured query plans, while a big data query plan can be an arbitrary DAG. We address the problem by converting an arbitrary DAG to a tree, as shown in Figure 12. We first consider a DAG as an "extended" tree by treating output operators (out degree = 0) as the "root(s)" and the input operators (in degree = 0) as the "leaves". Hence the operator dependencies point from a child node to a parent node. Consequently, the "extended" tree could have (i) nodes with multiple parents (multi-parent), and (ii) multiple "roots" (multi-root). When a node has multiple parents, we fork the subtree rooted on the node and link each subtree to one parent. When multiple "root" nodes are detected, we add one artificial node in the tree and point all the current "root" nodes to it, such that the artificial node becomes the only root for the transformed tree. After the transformations, the query plan could be converted into a tree and fits any tree-based model.
(2) How to model to leverage MCI features? QPPNet learns neural units in the plan structure and gets the latency prediction from the neural unit of the root operator. To leverage the instance-oriented features in channels 2-5, we broadcast them to all the neural units in the stage during preprocessing. Consequently, each neural unit can have both the operator-oriented features (Ch1+AIM) and the instance-oriented features to finish the instance latency prediction. Unlike QPPNet, TLSTM is quite compatible with our MCI framework after converting DAG plans to trees. The original TLSTM implicitly treats the hidden layer of the root operator as a representation of the query and learns the latency for it. In the extension, MCI+TLSTM applies the MCI framework with TreeLSTM as the plan embedder and the hidden layer of the root operator as the query embedding to learn the instance latency.

Customized Features in Figure 4

We will provide the list of customized features upon approval by Alibaba.

D MORE DETAILS ON IPA

D.1 Proof for Theorem 5.1

Denote \( m \) and \( n \) as the number of instances and machines, respectively. Denote \( L \in \mathbb{R}^{m \times n} \) as the latency matrix, \( L_{ij} \) as the \( i \)-th row in \( L \), and \( L_{ij}^\vartheta \) as the latency in the \( i \)-th row and the \( j \)-th column. And further denote IPA-L as the procedure of applying IPA over a latency matrix \( L \) and IPA-L as the stage latency achieved accordingly.

Recall that instances in a stage request the same CPU and memory when applying IPA and IPA can always find a feasible solution for a stage because the cluster’s capacity is always more than a stage’s need.

We consider each machine a 1-instance machine whose capacity can only afford one instance. Otherwise, we drop machines out of resource capacity and flat each \( k \)-instance machine (\( k \geq 1 \)) into \( k \) 1-instance machines. Since a stage can always find a feasible solution in the cluster, we have \( m \leq n \).

Under the column-order assumption C, let us denote \( \hat{L} \) as constructed by sorting rows of \( L \) in the non-ascending order such that \( \hat{L}_{1j} \geq \hat{L}_{2j} \geq ... \geq \hat{L}_{mj}, \forall j \).

**Proposition D.1.** Under the column-order assumption C, IPA-L and IPA-\( \hat{L} \) return the same placement plan and IPA-\( \hat{L} \) returns the instance-machine matches in the descending order of the rows.

**Proof.** Let us denote the machine sequence \( A = (a_1, a_2, ..., a_m) \) such that the best possible latency (BPL) for row 1, 2, ..., \( m \) in \( L \) is achieved at \( L_{1a_1}, L_{2a_2}, ..., L_{ma_m} \). Similarly, we denote the machine sequence \( \hat{A} = (\hat{a}_1, \hat{a}_2, ..., \hat{a}_m) \) as the machine sequence that BPLs are achieved for row 1, 2, ..., \( m \) in \( \hat{L} \). Since \( \hat{L} \) is constructed by sorting rows of \( L \), we have \( \hat{A} \) as one of the permutations of \( A \).

Notice that after each instance-machine match, IPA will disable the responding row and column in the latency matrix, and recursively choose the maximum (recalculated) BPL as the instance-machine match from the remaining latency matrix.

Consider after the \( t \)-th (\( 0 \leq t \leq m-1 \)) match, where we have the remaining latency matrices \( L, \hat{L} \in \mathbb{R}^{(m-t) \times (n-t)} \). Assume \((i, a_i)\) is the first instance-machine match picked by IPA in \( L \), such that \( \hat{L}_{ia_i} \) is the maximum among BPLs of all remaining rows. Since each entry in \( L \) and \( \hat{L} \) represents the running latency, we further assume each entry in \( L \) and \( \hat{L} \) has a unique value, and hence the column order in \( \hat{L} \) is descending.

Now we show that IPA always picks the BPL at the first row in \( \hat{L} \) at each step \( t \in [0, m-1] \) as the first instance-machine match by proving \( \hat{L}_{ia_i} = \hat{L}_{1a_i}, \) over the remaining latency matrices \( L \) and \( \hat{L} \).

When \( t = m-1 \), the remaining \( L \) has only one row, and hence \( \hat{L} = L \). Thus, they return the same BPL in the first and only row. When \( t < m-1 \), assume the first instance-machine match picked by IPA in \( \hat{L} \) is not in the first row. Without loss of generality, we assume the first match is \((2, \hat{a}_2)\) in \( \hat{L} \), i.e., \( \hat{L}_{2a_2} \) is the maximum among all BPLs from each row of \( \hat{L} \) and hence \( \hat{L}_{2a_2} \geq \hat{L}_{1a_1} \). Notice that by following the descending column order in \( \hat{L} \) at column \( a_1 \), we have \( \hat{L}_{2a_2} < \hat{L}_{1a_1} \). Since \((2, \hat{a}_2)\) achieves the BPL as the minimum latency in row \( \hat{L}_2 \), we have \( \hat{L}_{2a_2} < \hat{L}_{2a_2} \). Thus, we get \( \hat{L}_{2a_2} < \hat{L}_{1a_1} \), which contradicts our assumption that \( \hat{L}_{2a_2} \geq \hat{L}_{1a_1} \). Therefore, IPA picks \((1, a_1)\) as the first instance-machine match in \( \hat{L} \), which is originally from the first match \((i, a_i)\) in \( L \).

Therefore, IPA-\( \hat{L} \) always picks the same instance-machine match as IPA-L in each step, and hence IPA-\( \hat{L} \) and IPA-\( \hat{L} \) shall return the same placement plan. Since IPA-\( \hat{L} \) recursively picks the first row in the remaining \( \hat{L} \) in each step, we also show that IPA-\( \hat{L} \) returns the instance-machine matches following the descending order of the rows.

**Theorem D.1.** Under the column-order assumption C, IPA-\( \hat{L} \) achieves the minimum stage latency, and hence for IPA-\( \hat{L} \), such that \( IPA(L) = \sum_{m} \) l^m, where \( l^m \) denotes the groundtruth minimum stage latency of \( L \in \mathbb{R}^{m \times n} \).

**Proof.** According to Proposition D.1, we consider each column shares the same non-ascending order in \( L_{\text{mxn}} \). Given an arbitrary \( l_{\text{mxn}} \in \mathbb{R}^{m \times n} \) \( (m \leq n) \) that satisfies C, we show that IPA(L) achieves \( l^m_{\text{mxn}} \) by induction, where \( l^m_{\text{mxn}} \) is the ground truth minimum stage latency in \( L_{\text{mxn}} \).

**Base case.** When \( m = 1 \) and \( \forall n \geq 1 \), \( L \) has only one row and the stage latency equals the latency of the only instance. Notice IPA picks the machine that achieves the minimum latency for the first instance. Therefore, IPA(L) is also minimum achievable stage latency \( l_1^1 \). Thus, IPA(L) returns \( l^m_{\text{mxn}} \) for \( m = 1 \) and \( \forall n \geq 1 \).

**Induction.** Assume when \( m = k(k \geq 1) \) and \( \forall n \geq k \), IPA achieves the ground truth minimum stage latency, s.t., IPA(L) = \( l_k^k \). Given \( L_{(k+1) \times (n+1)} \in \mathbb{R}^{(k+1) \times (n+1)} \), we prove that IPA also achieves the minimum stage latency \( l^*_k \), s.t., IPA(L) = \( l^*_k \). In the following text, we use \( \hat{L} \) to represent IPA(L) for the simplicity of reading, and hence we are going to prove \( \hat{L} \) is \( l^*_{k+1} \).

Denote \( L_{ij} \) as the first matched latency by IPA and \( L_{kn} = L_{(k+1) \times (n+1)} \). Thus, \( L_{ij} \) is the maximum over \( L_{ij} \). Notice that IPA will continue to be applied on \( L_{kn} \). Hence we have \( \hat{L} = \max(L_{ij}, IPA(L_{kn})) \). Since \( L_{kn} \) has \( k \) rows and satisfies \( L \), IPA achieves the minimum stage latency IPA(L) = \( l^*_{k+1} \) according to the assumption. Therefore, we can rewrite \( \hat{L} = \max(L_{ij}, l^*_{k+1}) \).

Assume \( l^*_{k+1} \) is achieved by an optimal placement plan that provides a machine sequence \( B = (b_1, b_2, ..., b_{k+1}) \), s.t., instance 1, 2, ...
We get $k_{\hat{\beta}}$.

Case 3.

The minimum stage latency of the sequence and its ground truth minimum stage latency is $L_k$. By the definition of the stage latency, $L_k = \max_i s_{i,j}$. Notice that the first $i$ rows in $L_{k,n}$ are from row 2 to row $i$ in $L_{k+1, n+1}$. Since each column in $L_{k+1, n+1}$ keeps the non-ascending order, each of the first $i$ rows in $L_{k,n}$ is elementwisely less than or equal to that of $L'_{k,n}$. Therefore, we have $L_{k,n} = L'_{k,n}$.

Finally, $L_{k,n}$ is elementwisely less than or equal to $L'_{k,n}$. Since both latency matrices drop the column $j^*$ in $L_{k+1, n+1}$ and the last $k+1$ rows of the two matrices are both from row $i+1$ to row $k+1$ in $L_{k+1, n+1}$, the differences remain in the first $i$ rows. Notice that the first $i$ columns in $L_{k,n}$ are from row 2 to row $i$ in $L_{k+1, n+1}$, and the first $i$ columns in $L'_{k,n}$ are from row 1 to row $i$ in $L_{k+1, n+1}$. Since each column in $L_{k+1, n+1}$ keeps the non-ascending order, each of the first $i$ columns in $L_{k,n}$ is elementwisely less than or equal to that of $L'_{k,n}$, and hence $L_{k,n}$ is elementwisely less than or equal to $L'_{k,n}$.

Second, we prove $L_{k,n}^* \leq L'_{k,n}^*$ by contradiction. Assume $L_{k,n}^* < L_{k,n}^*$ and denote $B' = (b_{i,j}',...,b_{i,j})$ as the machine sequence that achieves $L_{k,n}^*$ in $L_{k,n}$, i.e., $B_{k,n}^* = \max_i L_{i,j}$. Since $L_{k,n}$ is elementwisely less than or equal to $L'_{k,n}$, we have $L_{k,n}^* = \max_i L_{i,j} \geq L_{i,j}'$. Therefore, $B'$ constructs one placement plan such that $B_{k,n}^* \leq L_{k,n}^*$. Furthermore, we have $I_{k,n}^* \leq I_{k,n}'^*$, according to the optimality of $B_{k,n}^*$. Thus, we have $I_{k,n}^* \leq L_{k,n}^* \leq L_{k,n}'$. Therefore, we have $L_{k,n}^* \leq L_{k,n}'^*$.

Recall that we have $I_{k,n}^* = \max L_{i,j}$, we construct the instance cluster $L_{i,j}' = \max L_{i,j}$. Notice that $L_{k,n}$ is the minimum latency picked over $k + 1$ rows. Therefore, we have $L_{k,n}^* = \max_i L_{i,j} = \max L_{i,j}$. Notice that $L_{k,n}^* = \max_i L_{i,j} = \max L_{i,j}$. Since $L_{k,n}^* = \max_i L_{i,j}$, and $L_{k,n}^* = \max_i L_{i,j}$, we have $L_{k,n}^* = \max_i L_{i,j}$.

Consider another machine sequence $B' = (b_{i,j}',...,b_{i,j})$ that picks machine $j^*$ for instance 1 and keeps the same machine choices for instance 2 to $k + 1$ as $B$. Since $L_{k,n}$ is the minimum latency of $L_{k,n}$, we have $L_{k,n}^* = L_{k,n}$. Notice that $L_{k,n}^* = \max i L_{i,j} = \max L_{i,j}$. Therefore, we have $L_{k,n}^* = \max_i L_{i,j} = \max L_{i,j}$. Since $L_{k,n}^* = \max_i L_{i,j}$, we have $L_{k,n}^* = \max_i L_{i,j}$.

Notice that $L_{k,n}^* = L_{k,n}'$ achieves the groundtruth minimum latency and picks $L_{i,j}'$. Therefore, $L_{k,n}^* = L_{k,n}'$.

Notice $B'$ achieves the groundtruth minimum latency and picks $L_{i,j}'$. Therefore, $L_{k,n}^* = L_{k,n}'$.

We now claim that $L_{k,n}^* \geq L_{k,n}'$, where $L_{k,n}'$ is the groundtruth minimum stage latency of $L_{k,n} = L_{k+1, n+1}/(L_1, L_j)$. Therefore, we have $L_{k,n}^* = L_{k,n}'$.

Therefore, we have $I_{k,n}' \leq I_{k,n}^*$. Hence we have $I_{k,n}' \leq I_{k,n}^*$.

D.2 Clustering-based IPA

By applying a clustering method, let us denote

1. $X' = [X_{1}',...,X_{m'}]$ as the $m'$ instance clusters
2. $Y' = [Y_{1}',...,Y_{n'}]$ as the $n'$ instance cluster representatives.
3. $\beta = [\beta_1,...,\beta_{m'}]$ denote the number of instances in each instance cluster.
4. $Y' = [Y_{1}',...,Y_{n'}]$ as the $n'$ machine clusters
5. $Y' = [y_1',...,y_{n'}]$ as the $n'$ machine cluster representatives.
6. $s = [s_1,...,s_{m'}]$ denotes the maximum instance each machine cluster can take.

Algorithm 4 shows the Pseudo code of the clustering based algorithm. We use a density-based clustering [1] that calculates the kernel density estimation (KDE) of the inputs in the 1D space and
uses the minima of the density distribution as the boundaries to divide instances into clusters.

It is worth mentioning that instances in the same cluster can still have smaller variances due to the differences in input row numbers. To avoid the case when the number of machines is not enough for running the entire instance cluster while the longer-running instances cannot be sent in line 9, our approach gives instances with a larger input row number a higher priority to be chosen in the same group. It also sorts the instances in the descending input row number order during clustering to reduce the time complexity during picking instances from the cluster.

E  MORE DETAILS ON RAA

E.1 Optimized Versions of RAA
Our Stage Optimizer supports several choices for using RAA.

Clustering choices. We provide choices for clustering including:

(1) RAA(W/O_C) does not cluster instances before applying our algorithm, which could achieve the best stage-level latency (exclusive solving time) and cost. However, it has a higher overhead than others.

(2) RAA(DBSCAN) applies a standard clustering approach, DBSCAN, to divide instances into several clusters. Clustering is performed over the MCI features of these instances.

(3) RAA(Fast_MCI) applies our customized MCI-based strategy to cluster instances. It leverages the instance clusters from IPA’s clustering result (based on fast 1D clustering) without additional time overhead. Specifically, we subdivide each instance cluster in IPA into separate new clusters based on the specific machine clusters that the different instances are sent. Notice that this process naturally happens at line 9 in Algorithm 4 when a subset of δ instances \((X^δ)\) from an instance cluster \((X^*_i)\) are sent to a subset of machines \((Y^δ)\) from the same machine cluster \((Y^*_i)\). Therefore it incurs no additional time overhead to generate new instance clusters and find the instance with the largest input row number as its representative.

Furthermore, when a clustering method is applied, each instance cluster would be reduced to one instance such that the latency of the instance cluster equals the latency of the representative instance and the cost of a cluster is the cost of the representative times the cluster size. After clustering, for ease of composition, we also use “instance” to refer to an instance cluster (when the clustering method is applied) when confusion does not arise.

By default, we choose RAA(Fast_MCI) for its efficiency.

MOO solutions. In this work, we provide two solutions to the hierarchical MOO problem:

(1) RAA(General) applies a general hierarchical MOO solution (Algorithm 2) for the general stage-level MOO problems.

(2) RAA(Path) applies a faster hierarchical MOO solution (Algorithm 3) for a specific case when the two objectives are stage-level latency and a summable cost metric (e.g., CPU-hours).

E.2 Additional Notation
Table 6 shows the notation for our description of the MOO solutions. Recall that a resource configuration (RC) refers to the configuration used for a container running on a specific instance, and it is an instance-specific configuration. A stage-level resource plan is a list of RC’s, one for each instance, and it is a stage-oriented solution. The design reserves the subscript exclusively for indicating the instances and uses superscripts to indicate others.

E.3 Proof of Proposition 5.1

Proposition 5.1 For a stage-level MOO problem, Algorithm 2 guarantees to find a subset of stage-level Pareto optimal points.

The corresponding notation of this proof is described in Table 6 and 7. First we give some details for the sub-procedure \(\text{find}_\text{optimal}\). The high level idea is that during the enumeration of the \(k_1\) objectives (that use the \(\max\) aggregate operator), once we have obtained some fixed values for the \(k_1\) objectives, we can determine the \(k_2\) objectives (that use the \(\sum\) aggregate operator) as follows: we transform the \(k_2\) objective values in each instance-level Pareto solution into a single value by utilizing a given weight vector. Then we select for each instance the instance-level solution \(f^*_j\) corresponding to the best weighted value, and sum up \(\{f^*_j, \ldots, f^*_m\}, i \in \{1 \ldots m\}\) to construct one stage-level Pareto solution, which contains the fixed \(k_1\) objective values and the summed values for the \(k_2\) objectives.

As a concrete example, suppose that we have two instances and three objectives, one using the \(\max\) aggregate operator and the other two using the \(\sum\) operator. For instance 1, there are two instance-level Pareto solutions \([15, 10, 5], [20, 15, 2]\). For instance 2, there are also two Pareto solutions, \([30, 5, 15], [40, 10, 5]\). As illustrated in Figure 7 and the example in Section 5.3, The function \(\text{find}_\text{all}_\text{possible}_\text{values}\) for objective 1 gives the list of possible values \([30, 40]\). Under the value 30 and the weight setting of \([0.5, 0.5]\) for the other two objectives, for instance 1, both solutions do not exceed 30, and the weighted sum is \(0.5 \times 10 + 0.5 \times 5 + 5 = 7.5\) (solution 1) and \(0.5 \times 15 + 0.5 \times 2 = 8.5\) (solution 2) respectively. So the solution \([15, 10, 5]\) is selected as the optimal for instance 1. In the instance 2, only the solution \([30, 5, 15]\) does not exceed 30, so \([30, 5, 15]\) is the optimal choice for instance 2. After that, a new stage-level solution is computed based on the selected instance-level solutions as, \([\max(15, 30) = 30, \sum(10, 5) = 15, \sum(5, 15) = 20]\). In the same way, under the value 40 of objective 1 and the weight setting of \([0.5, 0.5]\) for the other two objectives, the optimal choice for instance 1 and instance 2 are \([15, 10, 5]\) and \([40, 10, 5]\) respectively. We can obtain the stage-level values as \([\max(15, 40) = 40, \sum(10, 10) = 20, \sum(5, 5) = 10]\). So, the stage-level MOO set is \([30, 15, 20], [40, 20, 10]\). If we vary the weight vector as \([0.1, 0.9]\), we can find other stage-level MOO solutions, such as \([30, 20, 17]\) and \([40, 25, 7]\).

As a concrete example with more than one objectives using \(\max\) to see details in step 3 of Algorithm 2, suppose that we have two instances and four objectives, two (objective 1 and 2) using the \(\max\) aggregate operator and the other two (objective 3 and 4) using the \(\sum\) operator. For instance 1, there are two instance-level Pareto solutions \([15, 6, 10, 5], [20, 30, 15, 2]\). For instance 2, there are
also two Pareto solutions, [30, 10, 5, 15], [40, 50, 10, 5]. The function find_all_possible_values for objective 1 and 2 gives the lists of possible values [30, 40] and [10, 30, 50] respectively. Then, we use the the Cartesian product of these two lists as [[30, 10], [30, 30], [30, 50], [40, 10], [40, 30], [40, 50]]. Under the combination [30, 10], for instance 1, only the solution 1 does not exceed the objective 1 value 30 and the objective 2 value 10. So the solution [15, 6, 10, 5] is selected as the optimal for instance 1. In the instance 2, similarly, [30, 10, 5, 15] is the optimal choice for instance 2. After that, a new stage-level solution is computed based on the selected instance-level solutions as, [max(15, 30) = 30, max(6, 10) = 10, sum(10, 5) = 15, sum(5, 15) = 20]. In the same way, under the combination [30, 50] of objective 1 and 2, and the weight setting of [0.5, 0.5] for the objective 3 and 4, for instance 1, both solutions do not exceed the objective 1 value 30 and the objective 2 value 50, and the weighted sum is 0.5 * 10 + 0.5 * 5 = 7.5 (solution 1) and 0.5 * 15 + 0.5 * 2 = 8.5 (solution 2) respectively. So the solution [15, 6, 10, 5] with lower weighted sum is selected as the optimal for instance 1. In the instance 2, both solutions do not exceed the value of objective 2 (50), but only solution 1 does not exceed the value of objective 1 (30). Therefore, the solution 1 [30, 10, 5, 15] is selected as the optimal choice for instance 2. After that, a new stage-level solution is computed based on the selected instance-level solutions as, [max(15, 30) = 30, max(6, 10) = 10, sum(10, 5) = 15, sum(5, 15) = 20]. Then, after traversing all combinations of objective 1 and 2 under the weight setting of [0.5, 0.5], we get the stage-level MOO set [[30, 10, 15, 20], [40, 50, 20, 10]].

Now we first prove for a special case: when there are only k2 objectives using the sum operator.

**Lemma 1.** For a stage-level MOO problem with k2 objectives that use the sum operator only, Algorithm 2 guarantees to find a subset of stage-level Pareto optimal points.

In proving Lemma 1, we observe that Algorithm 2 is essentially a Weighted Sum procedure over Functions (WSF). Indeed we will prove the following two Propositions: 1) each solution returned

| Symbol | Description |
|--------|-------------|
| instance-oriented | k | the number of objectives. |
| | d | the number of resource types in a resource configuration (RC). |
| | m | the number of instances. |
| | i | the indicator to an instance. |
| | j | the indicator to a Pareto-optimal solution in a Pareto set (list). |
| | p_i | p_i is the number of Pareto-optimal solutions in the i-th instance. |
| | theta_i_j | the j-th RC corresponding to the Pareto-optimal solution for the i-th instance, theta_i_j \in \mathbb{R}^d. |
| | f_i_j | the j-th Pareto-optimal solution for the i-th instance. f_i_j \in \mathbb{R}^k. |
| | theta_i | the RC list, a list of RC corresponding to the Pareto set for the i-th instance. theta_i = [theta_i_1, theta_i_2, ..., theta_i_n]. |

| stage-oriented | theta | a list of RC list over m instances, theta = [theta_1, theta_2, ..., theta_m] |
| | f | a list of Pareto set over m instances, f = [f_1, f_2, ..., f_m] |
| | lambda_i | the index of the instance-level Pareto-optimal solution in the i-th instance. lambda_i \in [1, p_i] |
| | lambda | the index of the stage-level resources configuration based on theta, aka., the state. lambda = [lambda_1, lambda_2, ..., lambda_m] \in Lambda. |
| | pi | the step defined as to move a state lambda to another state lambda', such that lambda'_i = lambda_i + 1 and lambda'_j = lambda_j when j \neq i |
| | Theta^k | the stage-level resources configurations indicated by lambda, Theta^k = [Theta^k_1, Theta^k_2, ..., Theta^k_m]. |
| | Ps^k / F^lambda | the stage-level objective achieved by the stage-level resource configurations Theta^k, F^k \in \mathbb{R}^k. |
| | P0A | a list of resource plans corresponding to all the stage-level Pareto-optimal solutions. |
| | P0F | the Pareto set for the stage-level objectives, represented by a list of stage-level Pareto-optimal solutions. |

Table 6: Notation for the Hierarchical MOO problem
by WSF is Pareto optimal; 2) the solution returned by the function \(find\_optimal\) is equivalent to the solution returned by WSF. Then it follows that the solution returned by Algorithm 2 is Pareto optimal.

To introduce WSF, we first introduce the indicator function \(x_{ij}\), \(i \in [1, ..., m], j \in [1, ..., p_i]\), to indicate that the \(j\)-th solution in \(i\)-th instance is selected to contribute to the stage-level solution. \(\sum_{j=1}^{p_i} x_{ij} = 1\) means that only one solution is selected for each instance. Then \(x = [x_{1j}, ..., x_{mj}]\) represents the 0/1 selection for all \(m\) instances to construct a stage-level solution.

So for the \(v\)-th objective, its stage-level value could be represented as the function \(H\) applied to \(x\):

\[
F_v = H_v(x) = \sum_{i=1}^{m} \sum_{j=1}^{p_i} x_{ij} \cdot f_{ijv},
\]

where \(\sum_{j=1}^{p_i} x_{ij} = 1, i \in [1, ..., m], j \in [1, ..., p_i], v \in [1, ..., k_v]\)

Now we introduce the Weighted Sum over Functions (WSF) as:

\[
\text{argmin}_x \sum_{v=1}^{k_v} w_v \cdot H_v(x)
\]

s.t. \(\sum_{v=1}^{\alpha} w_v = 1\)

Next, we prove for Lemma 1. As stated before, It is done in two steps.

**Proposition E.1.** The solution constructed using \(x\) returned by WSF is Pareto optimal.

Proof. Assume that \(x^{*}\) (corresponding to \([F'_1, ..., F'_{k_v}]\)) is the solution of WSF. Suppose that an existing solution \([F''_1, ..., F''_{k_v}]\) (corresponding to \(x'\)) dominates \([F'_1, ..., F'_{k_v}]\). This means that \(\sum_{v=1}^{k_v} w_v \cdot H_v(x')\) is less than that of \(x^*\).

This contradicts that \(x^*\) is the solution of WSF. So there is no \([F''_1, ..., F''_{k_v}]\) dominating \([F'_1, ..., F'_{k_v}]\). Thus, \([F'_1, ..., F'_{k_v}]\) is Pareto optimal. \(\square\)

**Proposition E.2.** The optimal solution returned by the function \(find\_optimal\) in Algorithm 2 is equivalent to the solution constructed using \(x\) returned by WSF.

Proof. Suppose \(x'\) is returned by WSF. The corresponding stage-level solution is \([F'_1, ..., F'_{k_v}]\)

\[
x' = \text{argmin}_x \sum_{v=1}^{k_v} w_v \cdot H_v(x)
\]

\[
= \text{argmin}_x \sum_{v=1}^{k_v} w_v \cdot \left(\sum_{i=1}^{m} \sum_{j=1}^{p_i} x_{ij} \cdot f_{ijv}\right)
\]

\[
= \text{argmin}_x \sum_{i=1}^{m} \sum_{v=1}^{k_v} \sum_{j=1}^{p_i} (w_v \cdot f_{ijv} \cdot x_{ij})
\]

For the solution \([F''_1, ..., F''_{k_v}]\) returned by the function \(find\_optimal\) in Algorithm 2, \(x''\) represents the corresponding selection. It is achieved by minimizing the following formula:

\[
\sum_{i=1}^{m} (WS_{ij} \cdot x_{ij}) = \sum_{i=1}^{m} \sum_{v=1}^{k_v} (w_v \cdot f_{ijv} \cdot x_{ij})
\]

where \(WS_{ij} = \sum_{v=1}^{k_v} w_v \cdot f_{ijv} - \sum_{v=1}^{\alpha} w_v = 1\).

So, we have:

\[
x'' = \text{argmin}_x \sum_{i=1}^{m} \sum_{v=1}^{k_v} \sum_{j=1}^{p_i} (w_v \cdot f_{ijv} \cdot x_{ij})
\]

Since they are of the same form and achieve min at the same time, so we have \([F''_1, ..., F''_{k_v}] = [F'_1, ..., F'_{k_v}]\) \(\square\)

With these two propositions, we finish the proof of Lemma 1. Finally, we prove that for the general case that also involves \(k_1\) objectives that use the max aggregate operator, Proposition 5.1 holds as well.

Proof. Recall that Algorithm 2 enumerates all combinations of stage-level \(k_1\) objective values. Thus, there does not exist a (prefix of) stage-level \(k_1\) solution \([F'_1, ..., F'_{k_v}]\) that could not be found by algorithm 2. Putting it together with Lemma 1, we complete the proof of Proposition 5.1. \(\square\)

Here we discuss the time complexity when WS is utilized in the \(find\_optimal\) function in Algorithm 2. Recalling the Algorithm 2, it takes \(O(m \cdot \text{pmax})\) to find all the values within the lower and upper bounds of stage-level values for each max objectives. So, the Cartesian product of all \(k_1\) lists takes \(O((m \cdot \text{pmax})^{k_1})\). Within the loop of each combination, Algorithm 2 varies \(w\) weight vectors to generate multiple stage-level solutions. And under each weight vector, it takes \(O(m \cdot \text{pmax})\) to select the optimal solution for each instance based on WS. Thus, the overall time complexity is \(O((m \cdot \text{pmax})^{k_1} \cdot w \cdot (m \cdot \text{pmax})) = O((m \cdot \text{pmax})^{k_1 + 1})\).

For the particular case of \(k = 2\) with one using \(\max\) and the other using sum (the same case in Algorithm 3), \(k_1 = 1\) and \(k_2 = 1\), which means the weight vector for \(find\_optimal\) is fixed. Therefore, Algorithm 2 takes \(O((m \cdot \text{pmax})^{2})\).

**E.4 Proof of Proposition 5.2**

We show that Algorithm 3 guarantees to find the full set of stage-level Pareto-optimal points by induction. The notations in the proof are from Table 6 and 8.

Assume that the stage-level Pareto set \(P_{OF}\) has \(z\) solutions. Denote \(P_{OF} = [F_1^{(1)}, F_1^{(2)}, ..., F_1^{(z)}], P_{OF} = [\Phi^{(1)}, \Phi^{(2)}, ..., \Phi^{(z)}]\), and the state \(s^{(i)} \in \Lambda, i \in [1, z]\) indicates the \(i\)-th stage-level Pareto-optimal solution. After pre-sorting, we assume that both the stage-level and instance-level Pareto-optimal solutions are sorted
in the descending order of latency (and hence the ascending order of cost):

\[ U^{\lambda(1)} > U^{\lambda(2)} > \ldots > U^{\lambda(t)} \]

\[ O^{\lambda(1)} < O^{\lambda(2)} < \ldots < O^{\lambda(t)} \]

\[ u^i_1 > u^i_2 > \ldots > u^i_{p_i}, \forall i \in [1, m] \]

\[ o^i_1 < o^i_2 < \ldots < o^i_{p_i}, \forall i \in [1, m] \] (11)

To recall, Algorithm 3 maintains a max-heap \( \hat{Q} \) with \( m \) nodes built over the latencies among each of the \( m \) instances at a state \( \lambda \). At each RAA step from \( \lambda \) to \( \lambda' = \lambda + 1 \), \( \hat{Q} \) pops out the maximum latency with its corresponding instance id \( i \), and pushes back the \( (\lambda + 1) \)-th Pareto-optimal solution in instance \( i \). It stops the path when the \( (\lambda + 1) \)-th solution does not exist in the popped instance \( i = p_i \).

**Proof.** Now we show that Algorithm 3 covers \( \lambda^{(1)} \), ..., \( \lambda^{(z)} \) along the path.

**Base Case.** The state \( \lambda^{(1)} \) is in RAA path. When \( \lambda = [1, 1, \ldots, 1] \),

\[ O^{\lambda} = \sum_{i=1}^{1} (o^i_1) \leq \sum_{i=1}^{1} (o^i_1) \]

where “=” exists iff \( o^i_1 = o^i_1 \), \( \forall i \). Therefore, no other point in the stage-level objective space could dominate \( F^{\lambda} \) (on \( O^{\lambda} \)) and hence \( F^{\lambda} \) is a stage-level Pareto-optimal point. Notice

\[ U^{\lambda} = \max_{i \in [1, m]} (u^i_1) \geq \max_{i \in [1, m]} (u^i_1) \]

Therefore \( U^{\lambda} = \max_{\lambda \in \Lambda} U^{\lambda} \) is the maximum achievable stage latency and \( \lambda = \arg \max_{\lambda \in \Lambda} U^{\lambda} = \lambda^{(1)} \). Therefore \( \lambda^{(1)} \) is on the RAA path as the starting point.

**Induction:** Assume that state \( \lambda^{(t)}, t <= z - 1 \) is on the RAA path. Prove the state \( \lambda^{(t+1)} \) will also be on the RAA path.

Given \( \lambda^{(t)} \) indicating the \( t \)-th stage-level Pareto-optimal solution, we use \( O^* \) and \( U^* \) as shorthand for \( O^* = O^{\lambda^{(t)}}, U^* = U^{\lambda^{(t)}} \).

First, let us construct \( \hat{\lambda} \) using Eq. (12) and show that \( \lambda^{(t+1)} = \hat{\lambda} \).

\[ \hat{\lambda}_i = \begin{cases} 
\lambda^{(t)}_i + 1 & \text{if } u^{\lambda^{(t)}_i} = U^*, \\
\lambda^{(t)}_i & \text{otherwise } (u^{\lambda^{(t)}_i} < U^*), \forall i \in [1, m]. 
\end{cases} \] (12)

We prove two properties below.

(i) \( \hat{\lambda} \) is the minimum stage-level cost when the stage-level latency is smaller than \( U^* \). Notice for an instance \( i \) whose latency is not the largest at state \( \lambda^{(t)} \), the instance-level minimum cost has already been achieved at \( \hat{\lambda}_i = \lambda^{(t)}_i \) when the latency needs to be smaller than \( U^* \) according to the Pareto-optimality (otherwise, \( F^{\lambda^{(t)}} \) could have been dominated on the cost). For an instance \( i \) whose latency is equal to \( U^* \) at state \( \lambda^{(t)} \), \( u^{\lambda^{(t)}_i} = U^* \), its minimum cost shall be \( o^{\lambda^{(t)}_i} + 1 \) at \( \hat{\lambda}_i = \lambda^{(t)}_i + 1 \) when the latency needs to be smaller than \( U^* \), according to Eq. (11). Therefore, at state \( \hat{\lambda} \), the stage-level cost \( \hat{O}^{\lambda^{(t)}} \) is the summation of the minimum instance-level achievable cost among all instances when their latencies are smaller than \( U^* \).

(ii) \( \hat{\lambda} \) is the maximum stage-level latency that is smaller than \( U^* \). Otherwise, \( \exists \lambda' \in \Lambda, i \in [1, m], \text{such that } u^{\lambda'_{(i)}} < U^* \). According to the Pareto optimality in instance \( i \), we have the cost as \( o^{\lambda'_{(i)}} < o^{\lambda'_{(i)}} \). Hence, there exists a Pareto-optimal solution in instance \( i \) such that when the instance latency is smaller than \( U^* \), it achieves a smaller cost than the \( t \)-th stage-level optimal solution at state \( \lambda^{(t)} \), which violates the Pareto-optimality of \( F^{\lambda^{(t)}} \). Therefore, \( \hat{\lambda} \) is the maximum stage-level latency that is smaller than \( U^* \).

With (i) and (ii), we have \( \lambda^{(t+1)} = \hat{\lambda} \) as constructed.

Second, let us show that Algorithm 3 can reach \( \lambda^{(t+1)} \) by its policy. When \( \lambda^{(t)} \) is on the path, the maximum latencies in the max-heap \( \hat{Q} \) is \( U^* \). At each step, the policy will pop out an instance \( i \) whose latency is \( U^* \) achieved at \( \lambda^{(t)}_i \) and push back \( \lambda^{(t)}_i + 1 \). Therefore, each step addresses one instance that matches the first line of Eq. (12). Denote \( q \) as the number of instances that matches the first line of Eq. (12). Then we have \( \lambda^{(t+1)} \) on the path after \( q \) steps from \( \lambda^{(t)} \).

Therefore, we have all the stage-level Pareto-optimal solutions in the RAA path by using Algorithm 5.2.

**The time complexity** for sorting instance-level Pareto-optimal solutions costs \( O(m \cdot p_{\max} \log p_{\max}) \) over \( m \) instances. The algorithm uses \( O(m \log m) \) to build a max-heap. At each step, it includes one pop and one push, with \( O(\log m) \). It takes at most \( \sum_i p_i \) steps to iterates all instance-level Pareto-optimal solutions, which is \( O(\sum_i p_i) = O(m \cdot p_{\max}) \) iterations. Therefore the time complexity is \( O(m \cdot p_{\max} \log p_{\max} + \log m + m \cdot p_{\max} \log m) = O((1 + p_{\max})m \log m) = O(m \cdot p_{\max} \log(m \cdot p_{\max})) \).

---

| Symbol | Description |
|-------|-------------|
| \( u^i_1 \) | the latency in \( f^i_1 \). |
| \( o^i_1 \) | the cost in \( f^i_1 \). |
| \( u_i \) | a list of latency in \( f_i \): \( u_i = [u^i_1, u^i_2, \ldots, u^i_{p_i}] \). |
| \( o_i \) | a list of cost in \( f_i \): \( o_i = [o^i_1, o^i_2, \ldots, o^i_{p_i}] \). |
| \( U^{\lambda} \) | the stage-level latency achieved by the stage-level configurations \( \theta^{\lambda} \), \( U^{\lambda} = \max_{i \in [1,m]} (u^i_1) \in \mathbb{R} \). |
| \( O^{\lambda} \) | the stage-level cost achieved by the stage-level configurations \( \theta^{\lambda} \), \( O^{\lambda} = \sum_{i \in [1,m]} (o^i_1) \in \mathbb{R} \). |

**Table 8:** Additional notations for RAA path problem
F.4 Training Strategies and Evaluation

F.4.1 Training strategy. We employ two methods for training:

- **Retraining:** We retrain the model every 6 hours on the first day when data is insufficient and switch to the 24-hour retraining schedule from the second day.

- **Incremental Updates:** We also provide incremental updates by fine-tuning the model every 6 hours from the second day.

Before deploying the model online, we pick 16 hyperparameters as candidates by exploring the hyperparameter space intensively via the grid-search approach. Once deployed, we train models over the 16 hyperparameters simultaneously on 16 GPUs for every model update. The system switches to use the most recent model after a training or fine-tuning session finishes.

F.4.2 Training overhead. Our experiments involve traces of 0.62M jobs, 2M stages, 0.12B instances, and 407GB raw data over five consecutive days in three departments. The traces for training include:

- query plans from CBO;
- machine states and hardware information from the trace table at MaxCompute;
- the instance-level input cardinality of each stage from an in-memory distributed storage system.

The data preparation stage costs 30 minutes to generate the data for training by running ETL queries over those traces at MaxCompute with a parallelism of 219. Therefore, it costs ~2 minutes to collect and update the training data every day for a department.

F.3 Experimental Setup of Model Evaluation

Data Split and Training for the Model Analyzing. We follow the standard data split methods [??] used in the modern big data era, where the validation and test data set are representative of a much smaller portion (40K-95K) of the total (35M-50M). Figure 14 shows how we construct small and representative datasets for validation and testing over the three departments. Notice that the number of instances in a stage has huge skewness as shown in Figure 2(b). To address the issue, we count the frequency of each DAG structure from all instances, and divide the DAG structures in each workload into four categories by their frequency. We construct our test and validation set by applying different sampling strategies in each frequency type. For the high (>1K) and median (100-1K) type, we apply stratified sampling to pick 200 and 20 instances over each DAG structure; for the median-low (5-100) types, we randomly sample 20% instances per structure; on the remaining low frequency type, we randomly sample 20% over all instances. Each of our workloads has separate 40K-95K instances for validation and testing. And we use the remaining instances for training. We follow the common rules for training and hyperparameter tuning in the machine learning literature.

F.2 Simulation Environment

Hardware Property. Our experiments are deployed over 3 machines, each with 2x Intel Xeon Platinum 8163 CPU with 24 physical cores, 500G RAM, and 8 GeForce GTX 2080 GPU cards.

Simulator. Our simulator simulates the extended MaxCompute environment. First, to replace all the productive traces, it caches the query plans and instance meta information for the workload. Second, it reimplements the stage dependency manager and an extensible scheduler that can support both Fuxi and SO. Third, it generates the system states and capacity for each machine by sampling from prior knowledge. Lastly, it uses a Gaussian Processing Regression (GPR) model to sample the actual latency from a Gaussian distribution given the predicted latency. The GPR is pre-trained by learning the actual latency distribution against the latency predicted by a bootstrap model (by default, the MC1+GTN). During simulation, the GPR model takes a predicted latency and generates a Gaussian distribution \( N(\mu, \sigma) \) as the output. Then our simulator will sample from the distribution within \( \mu \pm 3\sigma \) to simulate the actual latency. We show the diagram of the simulator in Figure 16.

F.1 Workload Characteristics

We consider three representative workloads in a total of \( \approx 0.62 \) million jobs, with \( \approx 2 \) million stages and \( \approx 0.12 \) billion instances. Each workload consists of productive jobs from a business department over five consecutive days in Alibaba. As shown in Fig. 1, workloads A, B, and C show various properties:

1. The total number of jobs (41K-0.4M), stages (0.1M-1.0M), and instances (34M-50M).
2. The total size of collected traces (97GB-168GB).
3. The average number of stages per job (2.4-5.0).
4. The average number of instances per stage (85-1224).
5. The average number of operators per stage (3.7-6.3).
6. The average job latency (31-372s).
7. The average stage latency (15-182s).
8. The average instance latency (16-71s).

Workload A involves the most significant number of jobs (2-10x compared to B and C), while most are short-running jobs. Workload C has the least number of jobs but the longest job/stage/instance latencies, with its average job latencies 3-12.5x larger than A and B. It also has more instances on average, 12-14x times than A and B. Workload B has the most complex DAG topologies, with an average of 2x number of stages per job and an average of 1.2-1.7x number of operators per stage.

The statistics for data split in model training and evaluation are shown in Figure 14. We show the cumulative distribution functions for the metrics over the three projects in Figure 15.

F.2 Simulation Environment

Hardware Property. Our experiments are deployed over 3 machines, each with 2x Intel Xeon Platinum 8163 CPU with 24 physical cores, 500G RAM, and 8 GeForce GTX 2080 GPU cards.

Simulator. Our simulator simulates the extended MaxCompute environment. First, to replace all the productive traces, it caches the query plans and instance meta information for the workload. Second, it reimplements the stage dependency manager and an extensible scheduler that can support both Fuxi and SO. Third, it generates the system states and capacity for each machine by sampling from prior knowledge. Lastly, it uses a Gaussian Processing Regression (GPR) model to sample the actual latency from a Gaussian distribution given the predicted latency. The GPR is pre-trained by learning the actual latency distribution against the latency predicted by a bootstrap model (by default, the MC1+GTN). During simulation, the GPR model takes a predicted latency and generates a Gaussian distribution \( N(\mu, \sigma) \) as the output. Then our simulator will sample from the distribution within \( \mu \pm 3\sigma \) to simulate the actual latency. We show the diagram of the simulator in Figure 16.

F.3 Experimental Setup of Model Evaluation

Data Split and Training for the Model Analyzing. We follow the standard data split methods [??] used in the modern big data era, where the validation and test data set are representative of a much smaller portion (40K-95K) of the total (35M-50M). Figure 14 shows how we construct small and representative datasets for validation and testing over the three departments. Notice that the number of instances in a stage has huge skewness as shown in Figure 2(b). To address the issue, we count the frequency of each DAG structure from all instances, and divide the DAG structures in each workload into four categories by their frequency. We construct our test and validation set by applying different sampling strategies in each frequency type. For the high (>1K) and median (100-1K) type, we apply stratified sampling to pick 200 and 20 instances over each DAG structure; for the median-low (5-100) types, we randomly sample 20% instances per structure; on the remaining low frequency type, we randomly sample 20% over all instances. Each of our workloads has separate 40K-95K instances for validation and testing. And we use the remaining instances for training. We follow the common rules for training and hyperparameter tuning in the machine learning literature.

F.4 Training Strategies and Evaluation

F.4.1 Training strategy. We employ two methods for training:

- Retraining: We retrain the model every 6 hours on the first day when data is insufficient and switch to the 24-hour retraining schedule from the second day.

- Incremental Updates: We also provide incremental updates by fine-tuning the model every 6 hours from the second day.

Before deploying the model online, we pick 16 hyperparameters as candidates by exploring the hyperparameter space intensively via the grid-search approach. Once deployed, we train models over the 16 hyperparameters simultaneously on 16 GPUs for every model update. The system switches to use the most recent model after a training or fine-tuning session finishes.

F.4.2 Training overhead. Our experiments involve traces of 0.62M jobs, 2M stages, 0.12B instances, and 407GB raw data over five consecutive days in three departments. The traces for training include:

- query plans from CBO;
- machine states and hardware information from the trace table at MaxCompute;
- the instance-level input cardinality of each stage from an in-memory distributed storage system.

The data preparation stage costs 30 minutes to generate the data for training by running ETL queries over those traces at MaxCompute with a parallelism of 219. Therefore, it costs ~2 minutes to collect and update the training data every day for a department.

Figure 14: Data split for model evaluations

F. MORE ON PERFORMANCE EVALUATION

F.1 Workload Characteristics

We consider three representative workloads in a total of \( \approx 0.62 \) million jobs, with \( \approx 2 \) million stages and \( \approx 0.12 \) billion instances. Each workload consists of productive jobs from a business department over five consecutive days in Alibaba. As shown in Fig. 1, workloads A, B, and C show various properties:

1. The total number of jobs (41K-0.4M), stages (0.1M-1.0M), and instances (34M-50M).
2. The total size of collected traces (97GB-168GB).
3. The average number of stages per job (2.4-5.0).
4. The average number of instances per stage (85-1224).
5. The average number of operators per stage (3.7-6.3).
6. The average job latency (31-372s).
7. The average stage latency (15-182s).
8. The average instance latency (16-71s).

Workload A involves the most significant number of jobs (2-10x compared to B and C), while most are short-running jobs. Workload C has the least number of jobs but the longest job/stage/instance latencies, with its average job latencies 3-12.5x larger than A and B. It also has more instances on average, 12-14x times than A and B. Workload B has the most complex DAG topologies, with an average of 2x number of stages per job and an average of 1.2-1.7x number of operators per stage.

The statistics for data split in model training and evaluation are shown in Figure 14. We show the cumulative distribution functions for the metrics over the three projects in Figure 15.

F.2 Simulation Environment

Hardware Property. Our experiments are deployed over 3 machines, each with 2x Intel Xeon Platinum 8163 CPU with 24 physical cores, 500G RAM, and 8 GeForce GTX 2080 GPU cards.

Simulator. Our simulator simulates the extended MaxCompute environment. First, to replace all the productive traces, it caches the query plans and instance meta information for the workload. Second, it reimplements the stage dependency manager and an extensible scheduler that can support both Fuxi and SO. Third, it generates the system states and capacity for each machine by sampling from prior knowledge. Lastly, it uses a Gaussian Processing Regression (GPR) model to sample the actual latency from a Gaussian distribution given the predicted latency. The GPR is pre-trained by learning the actual latency distribution against the latency predicted by a bootstrap model (by default, the MC1+GTN). During simulation, the GPR model takes a predicted latency and generates a Gaussian distribution \( N(\mu, \sigma) \) as the output. Then our simulator will sample from the distribution within \( \mu \pm 3\sigma \) to simulate the actual latency. We show the diagram of the simulator in Figure 16.
In the training stage, we execute the periodical 24h retraining (over up to 50M instances) at midnight each day when the workloads are light, and the incremental fine-tuning (over ∼2M instances) every 6 hours. Figure 17 shows the training overhead of model updating over five consecutive days in three departments using 16 GPUs. The average training overhead for retraining is 3.5 hours, and the average training overhead for fine-tuning is 0.9 hour.

4.3 Modeling performance with the workload drift. To investigate the effects of the workload drift on the model performance, we design two experimental settings. One is the hypothetical worst, where we sorted stages in the descending order of the stage-level latency, divided them evenly into 120 hourly (5 day) workloads, and injected them into the system in this order. The other is more realistic, where we injected jobs by obeying the job arriving order in the real world. For each setting, we compare the following three training methods.

1. The static method: the model is retrained every 6 hours on the first 6 hours and never gets updated. Its WMAPE performance indicates the effect of workload drifts.
2. The retrain method: the model is retrained every 6 hours on the first day and gets periodically retrained every 24 hours from the second day.
Table 9: Modeling Performance for SiSL, SiOL, and ACTs

| Target | WL | WMAPE | MdErr | 95%Err | Corr | GlbErr |
|--------|----|-------|-------|--------|------|--------|
| SiSL   | A  | 8.6%  | 7.4%  | 62.4%  | 96.6%| 1.9%   |
|        | B  | 19.0% | 15.1% | 71.5%  | 96.4%| 5.4%   |
|        | C  | 15.1% | 12.7% | 97.3%  | 98.4%| 5.1%   |
| SiOL   | A  | 11.7% | 6.2%  | 52.4%  | 98.6%| 2.9%   |
|        | B  | 27.1% | 2.3%  | 46.5%  | 95.5%| 11.7%  |
|        | C  | 18.1% | 8.7%  | 65.2%  | 98.4%| 6.5%   |
| ACT    | A  | 6.6%  | 5.9%  | 38.0%  | 99.6%| 1.6%   |
|        | B  | 14.5% | 9.2%  | 40.6%  | 96.6%| 2.0%   |
|        | C  | 14.7% | 12.9% | 737.3% | 98.5%| 4.8%   |
| ACT*   | A  | 6.3%  | 5.3%  | 60.2%  | 90.1%| 0.9%   |
|        | B  | 12.5% | 7.8%  | 36.8%  | 97.3%| 0.9%   |
|        | C  | 10.9% | 9.7%  | 942.0% | 98.9%| 3.6%   |

Figure 17: Training overheads over 5 days

(3) The retrain+finetune method: the model has an incrementally fine-tuning every 6 hours from the second day in addition to the retrain method.

Setting 1. In the hypothetical worst case, the workload keeps drifting from the longest-running stage to the shortest-running stage over time. As shown in Fig. 18, the static method shows the worst performance, with the WMAPE increasing up to 10000%. The retrain and retrain+finetune methods both adapt to the workload drift, while the retrain+finetune method shows a better adaptivity with more incremental updates. As shown in Figure 18, the retrain+finetune method achieves an average of 6-10% WMAPE on the last day, while the retrain method gets 7-17%.

Setting 2. In the realistic setting, the workload pattern is defined as the mix of stages with short-, median- and long-running latencies on an hourly basis. The drifts are not monotonic and more complex. As shown in Fig. 20, the drift patterns are mainly based on 24 hours and the differences between hours vary from time to time. Therefore, if we only fine-tune every six hours, the retrain+finetune cannot adapt more to the hourly trend than the retrain method, as shown in Fig. 19. However, the retrain method can still catch the daily pattern and keep the average WMAPE low. For example, the hourly WMAPE in workload C mostly achieves as low as 15-25% over the last day, which performs much better than the average WMAPE of 831-5083% in the hypothetical worst setting.

F.5 Modeling Targets

We group the targets of the latency measurements in big data systems into the following four categories.

1. Single-instance Stage Latency (SiSL)
2. Single-instance Operator Latency (SiOL)
3. Multi-instance Stage Latency (MiSL)
4. Multi-instance Operator Latency (MiOL)

This paper aims at single-instance stage latency (SiSL) as the modeling target. Besides, we model SiOL for the breakdown error profiling and MiSL for performance comparison. MiOL is the modeling target used by CLEO [33] to evaluate the exclusive cost of a query operator.

F.6 Breakdown of Model Errors

Our breakdown analyses show that the errors are mainly from the IO-intensive operators (due to lack of features to capture IO/network activities in the current system) and the dynamic system states.

1. We train a separate model for SiOL by directly feeding GTN’s node embedding features to the MLP. Our tuned model achieves 12-27% WMAPEs and 2-8% MdErrs on SiOL over the three workloads, as shown in Table 9.
2. We average every operator’s latency and error across all instances, and observe that IO-intensive operators make up 59-84% of the WMAPE in the three workloads.
   - Workload A: 9.4%/11.7%
   - Workload B: 21%/27.1%
   - Workload C: 11%/18.1%
3. Figure 21 shows that the top-k error contributed operators make up at least 95% of the entire SiOL WMAPE in each workload. We sort the operators based on their average latencies in an instance and show the error rates of each operator in the x-axis ticks. We observe
   - IO-intensive operators take up most of the average latencies in an instance and account for most errors.
   - IO-intensive operators have more significant individual error rates. E.g., the StreamLineWrite (SLW) operator doing shuffle writing has the largest error rate in all workloads, with 17.9% in A, 55.4% in B, and 27.1% in C.
4. We look into 127 instances from the same stage with an SLW (StreamLineRead) operator in workload B. Besides the same query plan and resource plan, those instances run over machines with the same hardware types and similar system states with similar input row numbers for the SLW; however, the time costs for SLWs vary from 34s to 117s.
5. When we switch to a CPU-oriented metric actual CPU time (ACT) that measures the actual CPU cost in the lifetime of an instance, the WMAPE and MdErr reduce to 7-15% and 6-13%, respectively, as shown in Table 9.
Figure 18: Hourly WMAPE trends of three departments when injecting stages from long-running to short-running

Figure 19: Hourly WMAPE trends of three departments when injecting stages as the realistic setting
F.7 Impact of Discretized System States
Additional Expt 1: Impact of Discretized System States. We train separate models over system states with different discretized degrees (DD) and compare the model performances against optimization overhead in IPA. Figure 22 shows the examples in three workloads. When increasing DD, WMAPE converges to a certain range, while the complexity of the resource optimization problem is exponentially increased. WMAPE increases when DD is increased from 4 to 10 in Figures 22(b) and 22(c) due to the model overfitting. We choose DD=10 for workload A, and DD=4 for both B and C when doing resource optimization by exploring the DD for each workload.

F.8 Comparison of Modeling Targets
Additional Expt 2: Comparison of Modeling Targets. CLEO [33] learns a model for end-to-end query operator latency across multiple instances. While the CLEO code is not publicly available and we lack source data to train query operator latency, we construct an indirect comparison via modeling the end-to-end stage latency against our instance-level latency to illustrate the issues of CLEO’s choice of the modeling target. For the end-to-end stage latency, our best method can achieve only 37-54% WMAPE, much worse than 8.6-19% WMAPE of our instance latency model, showing that it is more challenging to predict end-to-end latency due to the high variability across different instances. The modeling performance of the end-to-end stage latency is in Table 10.

F.9 Subworkloads for RAA Evaluation
We record the trend of CPU utilization every day and average them over five consecutive days, where we slide a 40 min time window to find periods of the highest and the lowest average CPU utilization for each workload.

We subsample jobs on the two time periods for every workload on each of the five consecutive days to form the 29 subworkloads (workload C submitted 0 jobs during its idle period on the second day), including 12K jobs, 51K stages, and 6M instances.

F.10 More Results for Expt 6
We show more results on (1) stage-level latency reduction rates (both excluding and including the RO time overhead), (2) cloud cost reduction rates, (3) IPA time overhead for IPA(Org) and IPA(Cluster) in Figure 23 and Figure 24, respectively.

Fig. 24(a) shows that IPA reduces more stage latencies and cloud cost for workload C (with more long-running instances) than workload B (median-length stages) and workload A (short-running stages). Fig. 24(b) shows that IPA helps reduce more latency and cost when stages are submitted to an idle environment because the cluster can assign more machines of high capacity to a stage.

We also show the average stage latency, cloud cost and solving time in Table 11.

F.11 More Results for Expt 7
We show more results on (1) stage-level latency reduction rates (both excluding and including the RO time overhead), (2) cloud cost
Figure 22: Tradeoffs between WMAPE and the num. of system state combinations

| SO choice           | Coverage | $\bar{\text{Lat}}_{\text{stage}}^{(in)}$ (s) | $\text{Cost}_{\text{stage}}$ (0.001$)$ | $\bar{T}_{\text{stage}}$ (ms) | max($\bar{T}_{\text{stage}}$) (ms) |
|---------------------|----------|---------------------------------|---------------------------------|----------------------------|-------------------------------|
| FUXI                | 100%     | 100%                            | 100%                            | 100%                       | 100%                          |
| IPA(Org)            | 100%     | 100%                            | 100%                            | 100%                       | 100%                          |
| IPA(Cluster)        | 100%     | 100%                            | 100%                            | 100%                       | 100%                          |
| IPA+RAA(W/O_C)      | 100%     | 100%                            | 100%                            | 100%                       | 100%                          |
| IPA+RAA(DBSCAN)     | 100%     | 100%                            | 100%                            | 100%                       | 100%                          |
| IPA+RAA(Generic)    | 100%     | 100%                            | 100%                            | 100%                       | 100%                          |
| IPA+RAA(Path)       | 100%     | 100%                            | 100%                            | 100%                       | 100%                          |
| EVO                 | 0%       | 82%                            | 0%                              | 0%                         | 0%                            |
| WS(Sample)          | 90%      | 85%                            | 82%                             | 85%                        | 82%                           |
| PF(MOGD)            | 99%      | 98%                            | 98%                             | 98%                        | 98%                           |
| IPA+EVO             | 98%      | 100%                           | 98%                             | 100%                       | 98%                           |
| IPA+WS(Sample)      | 100%     | 100%                           | 100%                            | 100%                       | 100%                          |
| IPA+PF(MOGD)        | 100%     | 100%                           | 100%                            | 100%                       | 100%                          |

Table 11: Metrics of simulating the 29 subworkloads

Figure 23: Comparative results between IPA(Org) and Fuxi over 29 workloads

(a) reduction rates of $\bar{\text{Lat}}_{\text{stage}}^{(ex)}$, $\bar{\text{Lat}}_{\text{stage}}^{(in)}$, $\text{Cost}_{\text{stage}}$
(b) reduction rates of $\bar{\text{Lat}}_{\text{stage}}$ on busy/idle
(c) $\bar{T}_{\text{stage}}$ for IPA(Cluster)

Figure 24: Comparative results between IPA(Cluster) and Fuxi over 29 workloads

(a) reduction rates of $\bar{\text{Lat}}_{\text{stage}}^{(ex)}$, $\bar{\text{Lat}}_{\text{stage}}^{(in)}$, $\text{Cost}_{\text{stage}}$
(b) reduction rates of $\bar{\text{Lat}}_{\text{stage}}$ on busy/idle
(c) $\bar{T}_{\text{stage}}$ for IPA(Cluster)

reduction rates, (3) the solving time overhead for IPA+RAA(W/O_C), IPA+RAA(DBSCAN), IPA+RAA(Generic), IPA+RAA(Path) in Figure 25, 26, 27, and 28 respectively, with IPA(Cluster) as the default choice.
Among different workloads, IPA+RAA(Path) reduces a lot more cloud cost than IPA(Cluster) for workload C, which has more instances (505) per stage than B (42) and A (35), as shown in Fig. 28(a).

Among the busy and idle cluster environments, IPA+RAA(Path) shows a similar trend as IPA(Cluster) to reduce more latency and cost when a cluster is idle, as shown in Fig. 28(b).

We also show the average stage latency, cloud cost and solving time in Table 11.

F.12 More Analyses for Expt 8

We also show the average stage latency, cloud cost and solving time in Table 11 in the last 6 rows for EVO, WS(Sample), PF (MOGD), IPA+EVO, IPA+WS(Sample), and IPA+PF (MOGD).
when assuming the model is accurate. Our IPA+RAA dominates
To investigate the effects of IPA and RAA on the stages, we group all
Table 12 shows the model errors of instance latency and the cloud
1), IPA only (row 2), and IPA+RAA (row3). IPA reduces the stage
1.9M stages into short-, median- and long-running categories based
F.15 Diagnoses of Unexpected Behaviors
adjusts CPU cores and assigns x6 memory size for each instance
and short-running stages.
F.14 Breakdown Analyses on the End-to-End
Performance
To investigate the effects of IPA and RAA on the stages, we group all
1.9M stages into short-, median- and long-running categories based
on the default stage latencies and show the statistics in Table 13
when assuming the model is accurate. Our IPA+RAA dominates
the Fuxi policy in latency and cost on 93%-97% of the stages in each
category, with a 34-66% average latency reduction rate and 51-79%
average cost reduction rate. Specifically, our approach reduces more
latency and cost for long-running stages compared to the median-
and short-running stages.
Figure 29 shows the performances of the 479 instances (with
similar input cardinality) in a long-running stage, regarding their
latency, cost, cardinality, and resource plans. Figure 29(a) and 29(b)
show the latency and cost of each instance by using Fuxi (row 1), IPA
only (row 2), and IPA+RAA (row 3). IPA reduces the stage
latency by reducing the latency for 52% instances and increasing
the latency for the 34% instances. IPA+RAA reduces more latency
and cost after tuning the resource plan. In this example, IPA+RAA
adjusts CPU cores and assigns x6 memory size for each instance
and dominates the Fuxi policy in both latency and cost.
F.15 Diagnoses of Unexpected Behaviors
Ideally, the IPA+RAA approach should increase the resources for
long-running instances and decrease the resources for short-running
instances in a stage to make the instances more even and, in the
meantime, hit an instance-level Pareto-optimal point. While the
IPA+RAA approach manages to reduce the stage latency and cost
significantly, we observed the uneven behavior (Figure 30(a)) and
spikes (Figure 31(a)) in the instance-level latency distribution after
the optimization.
Now we enumerate the potential factors that cause the uneven
behavior and spikes.
First, the latency predictive model could have nonintuitive down-
linear behaviors. E.g., Figure 32 shows the predicted latency over
the CPU cores given a fixed memory size over several representative
instances in a stage. Notice when the requested CPU is less than
0.5, the predicted latency decreases by using fewer cores, which
is problematic. The improper modeling results underestimate
the latency of the short-running instances when the cores are small.
So we would miss some Pareto-optimal points and get the unexpected
behaviors. It turns out that although our data set is quite large, it
still does not have comprehensive coverage of the parameter space
on the resource plan. E.g, among the 34M instances in workload A,
we only observed 38 resource plans.
Second, we only consider the resource plan in a narrow range
due to the limited coverage of resource plan space in the training
data. The searching space of the resource plan is still in a nar-
row range even though being relaxed by assuming the choices of
requested cores and memory are independent in their min-max
ranges. However, when the resource plan is outside the searching
space, the model is not guaranteed to function properly, prevent-
ing us from finding better solutions. Specifically, we cannot lower
the cores anymore in the resource plan to enlarge the latencies of
the short-running instances (and hence could reduce the cost) to
make the distribution even. For example, the first 200 instances in
Figure 32(c) have already reached the lowest cores in the searching
space and cannot do things anymore.
Lastly, one may think our clustering could bring prediction errors
when assuming the latencies of the instances in a cluster are the
same, and hence contributes to the spikes. However, according
to our observations, the clustering works functionally well. E.g.,
Figure 33 shows the instance latency distributions of three stages,
where instances from the same cluster are given the same color.
In each stage, the latencies of the instances from the same cluster
are all quite close, verifying the good clustering performance. In
addition, it is worth mentioning that when we eliminate the factor
of the clustering, the issue of spikes could degenerate into an uneven
behavior, where instances of the spike latencies could belong to
one cluster of fewer members.
Therefore, our follow-up work will include (1) augmenting the
model with more offline sampling over the parameter space and (2)
taking the model uncertainty (the expectation and variance of the
latency prediction) into the concern.

| Bootstrap Model | Inst-latency (WMAPE) | Cost (GilErr) |
|-----------------|-----------------------|--------------|
| GTN+MCI         | 34%, 49%, 68%         | 48%, 41%, 71%|
| TLSTM           | 17%, 2%, 65%          | 43%, 31%, 70%|
| QPPNet          | 34%, 1%, 63%          | 46%, 30%, 68%|

Table 12: Predictive Errors VS Reduction Rate
Table 13: Breakdown analyses on the end-to-end effects

| Stages        | Short (<10s) | Median (10-100s) | Long (>100s) |
|---------------|--------------|------------------|--------------|
| % of stages that dominates the default RP in latency and cost | 68% | 99% | 97% |
| Avg latency reduction (s) | 0.9 | 11.7 | 201.8 |
| Avg latency reduction rate = (Avg latency reduction / Avg latency) | 46% | 54% | 65% |
| Avg cloud cost reduction | 0.002 | 0.019 | 1.697 |
| Avg cost reduction rate = (Avg cost reduction / Avg cost) | 77% | 62% | 75% |

(a) Instances latency in Fuxi, IPA and IPA+RAA
(b) Instances cost in Fuxi, IPA and IPA+RAA
(c) Instance cardinality, and resource plans

Figure 29: An example of stage breakdown analyses on the end-to-end performance

Figure 30: An example of uneven instance latency distribution
Instances latency in Fuxi, IPA and IPA+RAA

Instances cost in Fuxi, IPA and IPA+RAA

Instance cardinality, and resource plans

Figure 31: An example of instance latency spikes

Figure 32: Latencies distribution over different number of cpu cores

Figure 33: Examples of instance latencies in different instance clusters