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, et al.

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.

PVLDB Reference Format:
Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, Jingren Zhou. Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing. PVLDB, 15(11): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

1 INTRODUCTION

While big data query processing has become commonplace in enterprise businesses and many platforms have been developed for this purpose [3, 6, 7, 9, 13, 28, 31, 41, 48, 55, 59, 63, 64], resource optimization in large clusters [15, 34–36] 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 [28], 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 [63]. 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 [15, 18, 34, 35]. 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

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:XX.XX/XXX.XX

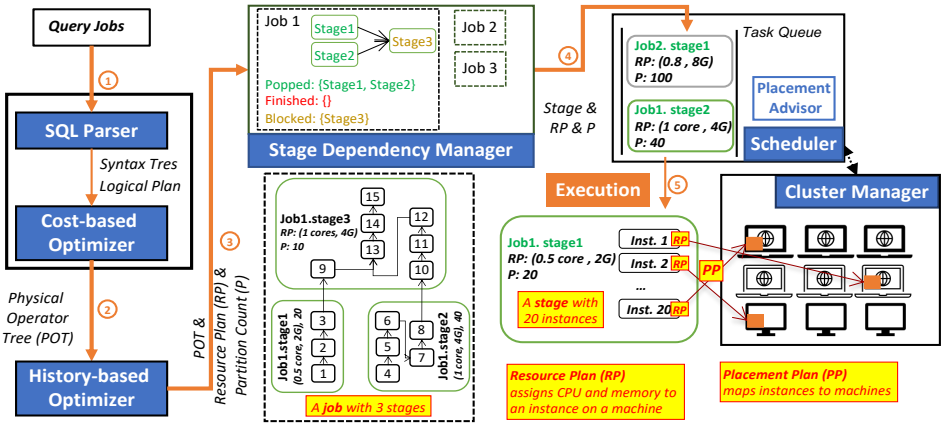


Figure 1: The lifecycle of a query job in MaxCompute

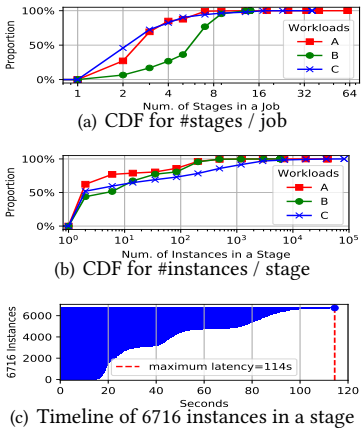


Figure 2: Trace overview

problem [15, 18] or the resource problem [36, 46] 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 [18] 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, 18, 19, 34, 35, 46, 61]. Only a few recent systems [36, 40] 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 [17, 18, 34] or machine learning methods [23, 26, 37, 65], have been used to improve SQL query plans [25, 45] or to improve selectivity estimation [11, 20, 21, 32, 33, 38, 50–54, 56, 57, 66]. 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 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 [26] 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 [37] 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 [23] 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 [65] 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 [28, 48, 59, 64].

ML-based models have been used for different purposes. NEO [25] 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. [45] train ML models from query logs to improve query plans for parametric queries. Phoebe [67] uses ML models for improving checkpointing decisions. Many recent works [11, 20, 21, 32, 33, 38, 50–54, 56, 57, 66] 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 [46, 61]). 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 [69] searches for good configurations by dividing high-dimensional configuration space into subspaces based on samples, but it cold-starts each tuning request. ClassyTune [68] solves the optimization problem by classification, which cannot be easily extended to the MOO setting. Among *learning-based* methods, Ottertune [46] 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 [61] 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 [62] 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 [63] and Yarn [48] make the scheduling decisions based on locality information, while Trident [13] 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 [18] 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 [18]), which is not affordable on today's productive scale (>10K machines). Morpheus [15] 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. PerfOrator [34] 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 [24] 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. [19] 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 [17] 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 [35, 49] 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.

UDAO [36, 60] 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 [40] 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 [14, 16, 42–44] 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 [16] 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* [27] is known to have *poor coverage* of the Pareto frontier [29]. *Normalized Constraints* [30] lacks in *efficiency* due to repeated recomputation to return more solutions. *Evolutionary Methods* [8] approximately compute a Pareto set but suffer from *inconsistent solutions*. *Multi-objective Bayesian Optimization* [5, 12] explores the potential Pareto-optimal points iteratively by extending the Bayesian approach, but lacks the *efficiency* due to a long-running time. *Progressive frontier* [36] 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 [28], 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) is a variant of the Cascades optimizer [10]. It 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 [63]. 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 [47], 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:

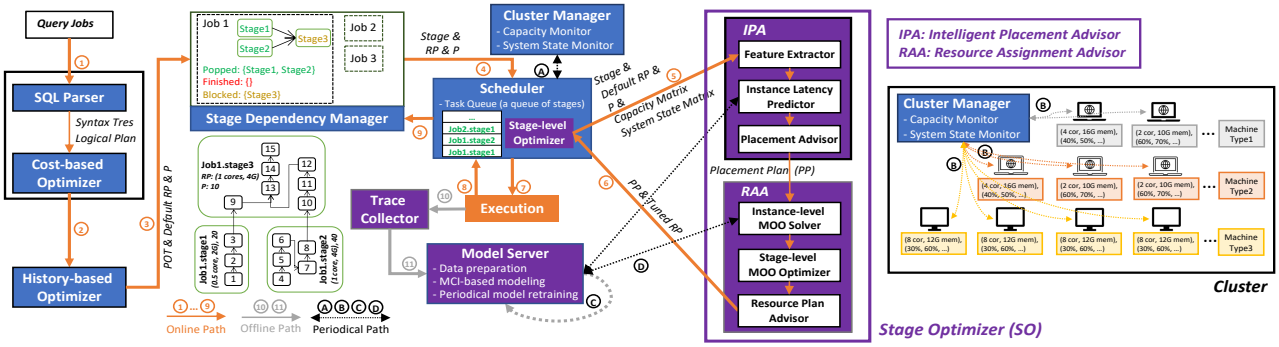


Figure 3: Extended system architecture for resource optimization

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 justifcation 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 or fine-tuning when the new traces are ready. It will serve as an instance-level latency predictor for resource optimization of online queries. **The Stage-level Optimizer** (SO) consists of the IPA and RAA, as described above. For a stage to be scheduled, it calls the predictive model to estimate the latency of each instance on any available machine, and determines the PP by minimizing stage latency and then the RP by minimizing both stage latency and cost.

4 FINE-GRAINED MODELING

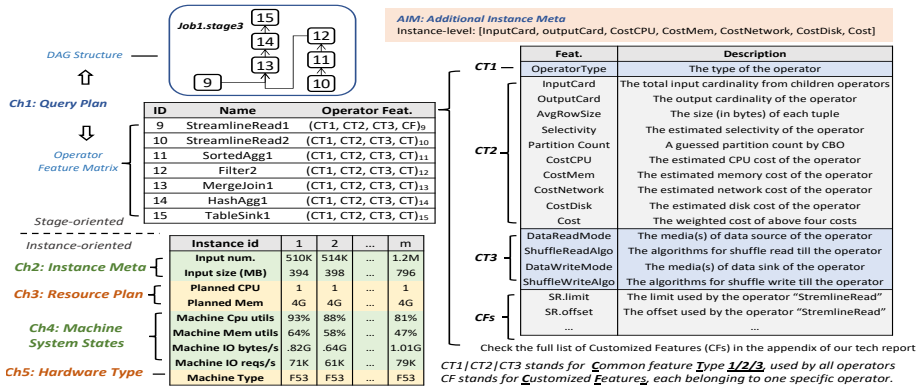
In this section, we present how to build fine-grained instance-level models that can be used to minimize both latency and cost of each stage. To suit the complexity of big data systems, our models capture all relevant systems aspects to support the resource optimizer to make effective recommendations.

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 coeffects 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



ID	Name	Operator Feat.
9	StreamlineRead1	(CT1, CT2, CT3, CF)$_9$
10	StreamlineRead2	(CT1, CT2, CT3, CT)$_{10}$
11	SortedAgg1	(CT1, CT2, CT3, CT)$_{11}$
12	Filter2	(CT1, CT2, CT3, CT)$_{12}$
13	MergeJoin1	(CT1, CT2, CT3, CT)$_{13}$
14	HashAgg1	(CT1, CT2, CT3, CT)$_{14}$
15	TableSink1	(CT1, CT2, CT3, CT)$_{15}$

Instance id	1	2	...	m
Input num.	510K	514K	...	1.2M
Input size (MB)	394	398	...	796
Planned CPU	1	1	...	1
Planned Mem	4G	4G	...	4G
Machine Cpu utils	93%	88%	...	81%
Machine Mem utils	64%	58%	...	47%
Machine IO bytes/s	.82G	.64G	...	1.01G
Machine IO reqs/s	71K	61K	...	79K
Machine Type	F53	F53	...	F53

Feat.	Description
OperatorType	The type of the operator
InputCard	The total input cardinality from children operators
OutputCard	The output cardinality of the operator
AvgRowSize	The size (in bytes) of each tuple
Selectivity	The estimated selectivity of the operator
Partition Count	A guessed partition count by CBO
CostCPU	The estimated CPU cost of the operator
CostMem	The estimated memory cost of the operator
CostNetwork	The estimated network cost of the operator
CostDisk	The estimated disk cost of the operator
Cost	The weighted cost of above four costs
DataReadMode	The media(s) of data source of the operator
ShuffleReadAlgo	The algorithms for shuffle read till the operator
DataWriteMode	The media(s) of data sink of the operator
ShuffleWriteAlgo	The algorithms for shuffle write till the operator
SR.limit	The limit used by the operator "StremlineRead"
SR.offset	The offset used by the operator "StremlineRead"
...	...

Figure 4: The multi-channel coverage from basic features

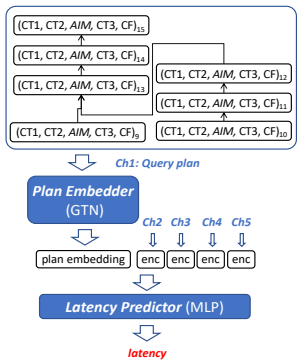


Figure 5: MCI-based modeling framework

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.* Channel 2-5 characterizes 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,* featurizes 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 (Ch1), even though their input row numbers can differ significantly. A model built on such features may have difficulty distinguishing those instance latencies.

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 instance-level input cardinality of operators 9 and 10 from Ch2, and calculate their output cardinality accordingly with the operator selectivity. Then, we derive the instance-level cardinality of the remaining operators according to the operator dependency. Finally, we calculate the instance-level operator costs 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 in a realistic setting.

Finally, recent ML models of DBMSs [26, 39] 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) [58] 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 [26] and TLSTM [37] are deferred to [22] .

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 [63] decides a PP for m instances as follows: (1) Identify the key resource (bottleneck) in the current cluster,

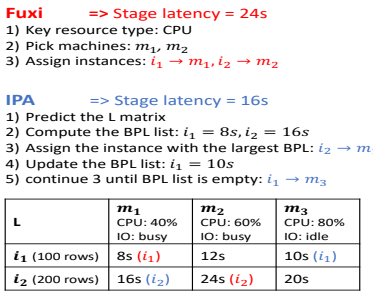


Figure 6: Example of IPA

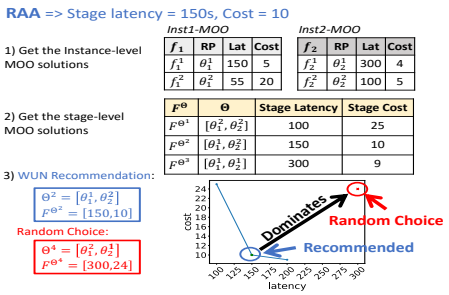


Figure 7: Example of RAA

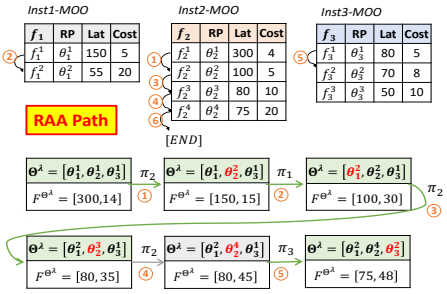


Figure 8: Example of RAA path

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 (i_1, i_2) to a cluster of 3 available machines (m_1, m_2, m_3). 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_1 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-specific resource 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 $f_1, ..., f_k$ to denote the set of predictive models of the k objectives and θ to denote a *resource configuration* available on that machine. Then the instance-level multiple-objective optimization (MOO) problem is defined as:

Definition 5.1. Multi-Objective Optimization (MOO).

$$\arg\min_{\theta} \quad f^{\theta} = f(\theta) = \begin{bmatrix} f_1(\theta) \\ \dots \\ f_k(\theta) \end{bmatrix}$$
$$s.t. \qquad \theta \in \Sigma \subseteq \mathbb{R}^d$$

where Σ denotes the set of all possible resource configurations.

A point $f' \in \mathbb{R}^k$ Pareto-dominates another point f'' iff $\forall i \in [1,k], f'_i \le f''_i$ and $\exists j \in [1,k], f'_j < f''_j$. A point f^* is **Pareto Optimal** iff there does not exist another point f' that Pareto-dominates it. Then, the **Pareto Set** $f = [f^{\theta^1}, f^{\theta^2}, ...]$ includes all the Pareto optimal points in the objective space $\Phi \subseteq \mathbb{R}^k$, under the configurations $[\theta^1, \theta^2, \dots]$, 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, $(x_1, ..., x_m)$, and n machines, $(y_1, ..., y_n)$. We use $\tilde{x}_i$ to denote the characteristics of x_i based on its features of Ch1 and Ch2 in its multi-channel representation, and $\tilde{y}_j$ to denote the features of Ch4 and Ch5 of machine y_j. Then consider two sets of variables, B and Θ:

(1) $B \in \mathbb{R}^{m \times n}$ is the binary assignment matrix, where $B_{i,j} = 1$ when x_i is assigned to y_j, and $\sum_j B_{i,j} = 1, \forall i = 1...m$.
(2) $\Theta = [\Theta_1, \Theta_2, ..., \Theta_m]$, denotes the collection of resource configurations of m instances, where $\forall i \in [1, \dots, m], \Theta_i \in \Sigma_i^* \subseteq \mathbb{R}^d$, and Σ_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., $f(\tilde{x}_i, \Theta_i, \tilde{y}_j)$ gives the latency when x_i is running on y_j using the resource configuration Θ_i. Then the stage-level latency can be written as, $L_{stage} = \max_{i,j} B_{i,j} f(\tilde{x}_i, \Theta_i, \tilde{y}_j)$. The stage cost is the weighted sum of cpu-hour and memory-hour and can be written as, $C_{stage} = \sum_{i,j} B_{i,j} f(\tilde{x}_i, \Theta_i, \tilde{y}_j)(\mathbf{w} \cdot \Theta_i^T)$, where $\mathbf{w}$ is the weight vector over d resources and $\mathbf{w} \cdot \Theta_i^T$ is the dot product between $\mathbf{w}$ and Θ_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.

$$\arg\min_{B,\Theta} \quad \begin{bmatrix} L(B,\Theta) = \max_{i,j} B_{i,j} f(\tilde{x}_i, \Theta_i, \tilde{y}_j) \\ C(B,\Theta) = \sum_{i,j} B_{i,j} f(\tilde{x}_i, \Theta_i, \tilde{y}_j)(\mathbf{w} \cdot \Theta_i^T) \\ \dots \end{bmatrix}$$
$$s.t. \quad \begin{array}{l} B_{i,j} \in \{0,1\}, \quad \forall i = 1...m, \forall j = 1...n \\ \sum_j B_{i,j} = 1, \quad \forall i = 1...m \\ \sum_i B_{i,j} \Theta_i^1 \le U_j^1, \ \dots, \ \sum_i B_{i,j} \Theta_i^d \le U_j^d \ , \forall j = 1...n \end{array}$$

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 [8, 27, 30, 36] 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's of thousands. Hence, both $B \in \mathbb{R}^{m \times n}$ and $\Theta = [\Theta_1, \Theta_2, ..., \Theta_m], \forall i, \Theta_i \in \mathbb{R}^d$, 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 Θ_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 Def. 5.2 by treating Θ_0 as a constant.

Step 2 (RAA): Given the solution from step 1, B^*, we now minimize over the variables Θ 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 Θ_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 $L_{i,j} = f(\tilde{x}_i, \Theta_0, \tilde{y}_j)$, ignoring the constant Θ_0, and focusing on the stage latency objective in Def. 5.2, we obtain:

$$\arg\min_{B} \Big[\ L(B) = \max_{i,j} B_{i,j} L_{i,j}\ \Big] \qquad (3)$$
$$\text{s.t.} \quad \sum_j B_{i,j} = 1, \forall i \ \text{ and } \ \sum_i B_{i,j} \le \beta_j, \forall j$$

Here, we add the constraint $\sum_i B_{i,j} \le \beta_j$ 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 β_j denote the maximum number of instances a machine y_j can take based on its capacity constraint ($U_j \in \mathbb{R}^d$) and diversity preference. We have

$$\beta_j = \min\{\lfloor U_j^1 / \Theta_0^1 \rfloor, \lfloor U_j^2 / \Theta_0^2 \rfloor, ..., \lfloor U_j^d / \Theta_0^d \rfloor, \alpha\}$$

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

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.

[1]Otherwise, there is a scheduling delay for the stage with the admission control.

Algorithm 1: IPA Approach

```
L = cal_latency(model, X, Y), S = cal_max_num_inst().
Y* = Y, X* = X, and P = {}.                                   // Init
BPL_list = cal_bpl(L, X*, Y*).
repeat
    i_t, j_t = argmax(BPL_list)
        // get the instance and machine index pairs of the largest BPL
    P = P ⋃ {x_{i_t} → y_{j_t}}, X* = X* − {x_{i_t}}, S_{j_t} = S_{j_t} − 1.
    if Y* is ∅ and X* is not ∅ then
        return {}                                             // No solution found
    end if
    if S_{j_t} == 0 then Y* = Y* − {y_{j_t}}, Recalculate the BPL_list.
    end if
until X* is empty
return P
```

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 $O(m(m+n)+d)$ 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, Θ_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 [22] 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, 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 $O(m \log m + n \log n + m'(m' + n') + d)$, 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 [18] 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:

$$\arg\min_{\Theta} \; F^{\Theta} = F(\Theta) = \begin{bmatrix} F_1(\Theta) = g_1(f_1(\Theta_1), ..., f_1(\Theta_m)) \\ ... \\ F_k(\Theta) = g_k(f_k(\Theta_1), ..., f_k(\Theta_m)) \end{bmatrix} \quad (4)$$

where $\Theta = [\Theta_1, \Theta_2, ..., \Theta_m] = [\theta_1^{l_1}, \theta_2^{l_2}, ...\theta_m^{l_m}]$, $\theta_i^{l_i}$ denotes the l_i-th resource configuration of the i-th instance ($i \in \{1, ..., m\}$), and the aggregator g_j (j=1,...,k) is either a sum or max.

As in the instance-level, we define the **stage-level resource configuration** as $\Theta = [\theta_1^{l_1}, \theta_2^{l_2}, ...\theta_m^{l_m}]$, with its corresponding F^{Θ} in the objective space $\Phi \subseteq \mathbb{R}^k$. 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 = max) and f_2 for cost (g_2 = sum). Suppose $\Theta = [\theta_1^1, \theta_2^2]$ as the stage-level resource configuration. Then we have the stage-level latency as $F_1(\Theta) = \max(150, 100) = 150$, the stage-level cost as $F_2(\Theta) = sum(5, 5) = 10$, and the stage-level solution F^{Θ} 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.

Algorithm 2: General Hierarchical MOO

Require: f_i^j, $i \in [1, m]$, $j \in [1, p_i]$, where p_i is the number of Pareto-optimal solutions in i-th instance and $[\theta_i^1, ..., \theta_i^{p_i}]$

1: $PO_\Theta = []$, $PO_F = []$
2: minMList, maxMList = *find_range*(f)
3: k1Combs = *find_all_possible_values*(f, minMList, maxMList)
4: **for** c **in** k1Combs **do**
5: **for** i **in** m **do**
6: optimal_solution, index = *find_optimal*(c, $[f_i^1, ...f_i^{p_i}]$)
7: $\Theta_i = \theta_i^{index}$
8: **end for**
9: $PO_\Theta.append(\Theta)$, $PO_F.append(F(\Theta))$
10: **end for**
11: PO_F, PO_Θ = *filter_dominated*(PO_F, PO_Θ)
12: **return** PO_F, PO_Θ

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 [36] 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 θ_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 (*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 can not afford the enumeration, which grows exponentially in the number of instances, $O(p_{max}^m)$ 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 *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 Θ choice ($\Theta = [\theta_1^2, \theta_2^2]$) 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 $\Theta = [\theta_1^2, \theta_2^1]$

Algorithm 3: RAA Path policy to construct PO_Θ and PO_F

Require: p_i, θ_i^j, f_i^j, $\forall i \in [1, m]$, $j \in [1, p_i]$, where we pre-sort f_i^j and θ_i^j in the descending order of latency for each instance.

1: $u_i^j = lat(f_i^j)$, $\forall i \in [1, m]$, $j \in [1, p_i]$.
2: $PO_\Theta = []$, $PO_F = []$, $\lambda = [1, 1, \ldots, 1]$, smax=inf
3: Build the max heap Q with data points $[(u_i^1, i)]$, $\forall i \in [1, m]$.
4: **repeat**
5: qmax, $i = Q.pop()$
6: **if** qmax < smax **then**
7: $PO_\Theta.append(\Theta^\lambda)$, $PO_F.append(F(\Theta^\lambda))$, smax = qmax
8: **end if**
9: $\lambda = \pi_i(\lambda)$
10: **if** $\lambda_i > p_i$ **then return** PO_Θ, PO_F
11: **end if**
12: $Q.push((u_i^{\lambda_i}, i))$
13: **until** True

and $[300, 9]$ with $\Theta = [\theta_1^1, \theta_2^1]$. 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.

Fast hierarchical MOO solution: RAA Path. In the case that there are only two objectives, whose aggregators are `max` and `sum`, respectively, we provide another algorithm that can resolve the Hierarchical MOO efficiently. The key idea here is that the two objectives are indeed making tradeoffs. The increase in one objective often leads to a decrease in the other. With this observation, we design a new approach called "RAA path".

Example. Figure 8 shows an example of the Pareto sets of 3 instances in a stage, where each instance has 2, 4, 3 Pareto-optimal solutions, and each solution is sorted by latency in descending order. We start the procedure by selecting the first Pareto solution for each instance, this leads to $\Theta^\lambda = [\theta_1^1, \theta_2^1, \theta_3^1]$ and $F^\lambda = [300, 14]$. Notice that 300 is corresponding to θ_2^1, so we replace θ_2^1 by the entry below it in the same Pareto set. Now we have the second solution as $\Theta^\lambda = [\theta_1^1, \theta_2^2, \theta_3^1]$ and $F^\lambda = [150, 15]$. Continue in this manner; each time, select the θ corresponding to the instance of the max latency and replace it with the item below it in the same Pareto set. It terminates until there is no entry below the current θ.

For a formal description, we use the following notation: (1) $\lambda = [\lambda_1, \lambda_2, ..., \lambda_m]$ as a state, where λ_i is the index of the Pareto point in instance i with the λ_i-largest latency; (2) $\Theta^\lambda = [\theta_1^{\lambda_1}, \theta_2^{\lambda_2}, ..., \theta_m^{\lambda_m}]$; (3) $\pi_i : \lambda \rightarrow \lambda'$ is a *step* such that the i^{th} dimension in state λ is increased by 1. (4) p_i is the number of instance-level Pareto solutions for instance i. 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, RAA path guarantees to find the full set of stage-level Pareto optimal points at the complexity of $O(m \cdot p_{max} \log(m \cdot p_{max}))$.

RAA with Clustering. For efficiency, we run both the General hierarchical MOO and RAA Path methods by clustering the instances and machines, where m is replaced by $m' << m$ in the complexity.

WL	Num. Jobs	Num. stages	Num. Inst	#stages /job	#insts /stage	#ops /stage	Avg Job Lat(s)	Avg Stage Lat(s)	Avg Inst Lat(s)
A	405K	970K	34M	2.40	35.45	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	505.51	5.31	376.83	181.88	71.08

Table 1: Workload statistics for 3 workload over 5 days

Resource plan recommendation. After getting the stage-level Pareto set, we reuse UDAO's Weighted Utopia Nearest (WUN) strategy to recommend the resource plan, which includes a configuration for each instance of the stage. It recommends the resource plan whose objectives could achieve the smallest distance to the Utopia point, which is the hypothetical optimal in all objectives.

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 [22]) by replaying the production traces.

Workload Characteristics. See Table 1 for descriptions of production workloads A-C. We give additional details in [22].

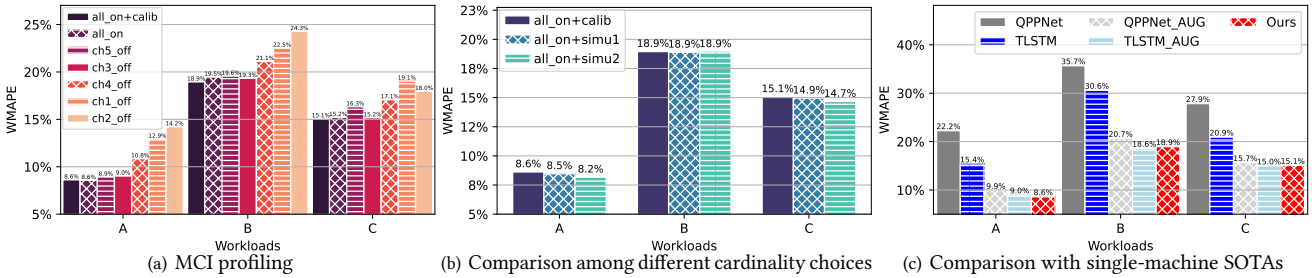
6.1 Model Evaluation

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+GTN over all channels augmented by AIM for the instance latency prediction. More details about training are given in [22].

We report model accuray on 5 metrics: (1) weighted mean absolute percentage error (WMAPE), (2) median error (MdErr), (3) 95 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 of more importance 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 3-4.5x 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



(a) MCI profiling (b) Comparison among different cardinality choices (c) Comparison with single-machine SOTAs

Figure 9: Performance of our instance-level models, compared to the state-of-the-art (SOTA) methods

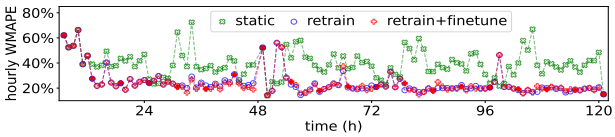


Figure 10: WMAPEs over time (workload C)

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 [22].

Expt 2: *Multi-channel Inputs.* We now investigate the importance of each channel to model performance. We train separate models with different input channel choices, including (1) the leave-one-out choices for a channel `x` (`Chx_off`), (2) the five basic channels `all_on`, and (3) the five basic channels and the AIM `all_on+calib`.

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` applies 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% WAMPE 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 [35].

Expt 4: *Comparison with QPPNet* [26] *and TLSTM* [37]. 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 embedder). 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 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. (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 in [22].

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 [63], 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 [22]).

We consider the following metrics in resource optimization (RO): (1) *coverage*, the ratio of stages that receive feasible solutions within 60s; (2) $Lat_s^{(in)}$, the average stage latency that includes the RO time; (3) $Cost_s$, the average cloud cost of all stages in a workload; (4) T_s, 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.

SO choice	Coverage A	Coverage B	Coverage C	$Lat_s^{(in)}$ ↓ A	$Lat_s^{(in)}$ ↓ B	$Lat_s^{(in)}$ ↓ C	$Cost_s$ ↓ A	$Cost_s$ ↓ B	$Cost_s$ ↓ C	$avg(T_s)$ (ms) / $max(T_s)$ (ms) A	B	C
IPA(Org)	100%	100%	100%	11%	20%	51%	5%	9%	15%	280 / 2.0K	17 / 18	1.4K / 1.8K
IPA(Cluster)	100%	100%	100%	**12%**	**17%**	**50%**	**4%**	**7%**	**14%**	**15 / 24**	**10 / 10**	**33 / 36**
IPA+RAA(W/O_C)	100%	100%	100%	8%	79%	58%	31%	76%	75%	2.5K / 19K	177 / 220	3.5K / 6.3K
IPA+RAA(DBSCAN)	100%	100%	100%	27%	69%	67%	21%	64%	74%	223 / 937	132 / 136	258 / 452
IPA+RAA(General)	100%	100%	100%	36%	80%	76%	29%	75%	75%	100 / 241	20 / 23	167 / 229
IPA+RAA(Path)	100%	100%	100%	**36%**	**80%**	**76%**	**29%**	**75%**	**75%**	**98 / 226**	**17 / 18**	**156 / 224**
EVO	0%	82%	0%	−%	-36%	−%	−%	66%	−%	− / −	21K / 24K	− / −
WS(Sample)	90%	85%	82%	-140%	48%	-74%	-107%	49%	-52%	7.4K / 22K	465 / 753	9.8K / 12K
PF(MOGD)	99%	100%	98%	-15%	49%	65%	24%	56%	75%	2.7K / 4.0K	2.1K / 2.5K	1.5K / 2.3K
IPA+EVO	98%	100%	98%	-27%	69%	43%	22%	75%	71%	3.0K / 7.3K	2.4K / 2.6K	5.0K / 5.9K
IPA+WS(Sample)	100%	100%	100%	-36%	76%	-1%	-19%	72%	32%	3.5K / 10K	517 / 741	12K / 18K
IPA+PF(MOGD)	100%	100%	100%	-0.4%	51%	69%	26%	56%	75%	1.6K / 2.5K	1.2K / 1.6K	1.2K / 2.2K

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

WL	WMAPE	MdErr	95%Err	Corr	GlbErr
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%

Table 3: Modeling Performance

SO scenario	Stage Lat (in):	Cost:
IPA (noise-free)	**10%, 15%, 44%**	**3%, 7%, 12%**
IPA (noisy)	**9%, 10%, 42%**	**3%, 6%, 12%**
IPA+RAA (noise-free)	**37%, 58%, 72%**	**43%, 56%, 78%**
IPA+RAA (noisy)	**37%, 55%, 72%**	**42%, 56%, 78%**
Bootstrap Model	Stage Lat (in):	Cost:
GTN+MCI	34%,49%,68%	48%,41%,71%
TLSTM	**17%**,**2%**,65%	43%,**31%**,70%
QPPNet	34%,**1%**,63%	46%,**30%**,68%

Table 4: Average RR over 2M stages

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 2-83x 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 IPA). IPA+RAA(W/O_C) does RAA without clustering and suffers high 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(General) shows the performance of our general hierarchical MOO approach, which is slightly worse than IPA+RAA(Path) (in this 2D MOO problem) in running time while offering a similar reduction of latency and cost. Details of the four choices are in [22].

Expt 8: *MOO baselines.* We next compare to SOTA MOO solutions, EVO [8], WS(Sample) [27], and PF(MOGD) [36], using Def. 5.2. See [22] 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 them 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 Θ 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 $\mu \pm 3\sigma$. Table 4 shows that in both noise-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.

7 CONCLUSIONS

We presented a MaxCompute [28] based big data system that supports multi-objective resource optimization via fine-grained instance-level modeling and optimization. To suit the complexity of our system, we developed fine-grained instance-level models that encode all relevant information as multi-channel inputs to deep neural networks. By exploiting these models, our stage optimizer employs a new IPA module to derive a latency-aware placement plan to reduce the stage latency, and a novel RAA model to derive instance-specific resource plans to further reduce stage latency and cost in a hierarchical MOO framework. Evaluation using production workloads shows that (1) our best model achieved 7-15% median error and 9-19% weighted mean absolute percentage error; (2) compared to the Fuxi scheduler [63], IPA+RAA achieved the reduction of 37-72% latency and 43-78% cost while running in 0.02-0.23s.

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.

REFERENCES

[1] Charu C. Aggarwal and Chandan K. Reddy (Eds.). 2014. *Data Clustering: Algorithms and Applications*. CRC Press. http://www.crcpress.com/product/isbn/9781466558212

[2] Malay Bag, Alekh Jindal, and Hiren Patel. 2020. Towards Plan-aware Resource Allocation in Serverless Query Processing. In *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*, Amar Phanishayee and Ryan Stutsman (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud20/presentation/bag

[3] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*. 1151–1162.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[5] Samuel Daulton, Maximilian Balandat, and Eytan Bakshy. 2020. Differentiable Expected Hypervolume Improvement for Parallel Multi-Objective Bayesian Optimization. *CoRR* abs/2006.05078 (2020). arXiv:2006.05078 https://arxiv.org/abs/2006.05078

[6] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* (San Francisco, CA). USENIX Association, Berkeley, CA, USA, 10–10.

[7] Sergey Dudoladov, Chen Xu, Sebastian Schelter, Asterios Katsifodimos, Stephan Ewen, Kostas Tzoumas, and Volker Markl. 2015. Optimistic Recovery for Iterative Dataflows in Action. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1439–1443. https://doi.org/10.1145/2723372.2735372

[8] Michael T. Emmerich and André H. Deutz. 2018. A Tutorial on Multiobjective Optimization: Fundamentals and Evolutionary Methods. *Natural Computing: an international journal* 17, 3 (Sept. 2018), 585–609. https://doi.org/10.1007/s11047-018-9685-y

[9] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. 2009. Building a HighLevel Dataflow System on top of MapReduce: The Pig Experience. *PVLDB* 2, 2 (2009), 1414–1425.

[10] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29. http://sites.computer.org/debull/95SEP-CD.pdf

[11] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1035–1050. https://doi.org/10.1145/3318464.3389741

[12] Daniel Hernández-Lobato, José Miguel Hernández-Lobato, Amar Shah, and Ryan P. Adams. 2016. Predictive Entropy Search for Multi-objective Bayesian Optimization. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. JMLR.org, 1492–1501. http://proceedings.mlr.press/v48/hernandez-lobatoa16.html

[13] Herodotos Herodotou and Elena Kakoulli. 2021. Trident: Task Scheduling over Tiered Storage Systems in Big Data Platforms. *Proc. VLDB Endow.* 14, 9 (2021), 1570–1582. http://www.vldb.org/pvldb/vol14/p1570-herodotou.pdf

[14] Arvind Hulgeri and S. Sudarshan. 2002. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong, China) *(VLDB '02)*. VLDB Endowment, 167–178. http://dl.acm.org/citation.cfm?id=1287369.1287385

[15] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Iñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 117–134. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/jyothi

[16] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis Ioannidis. 2011. Schedule Optimization for Data Processing Flows on the Cloud. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. ACM, New York, NY, USA, 289–300. https://doi.org/10.1145/1989323.1989355

[17] Viktor Leis and Maximilian Kuschewski. 2021. Towards Cost-Optimal Query Processing in the Cloud. *Proc. VLDB Endow.* 14, 9 (2021), 1606–1612. http://www.vldb.org/pvldb/vol14/p1606-leis.pdf

[18] Jiexing Li, Jeffrey F. Naughton, and Rimma V. Nehme. 2014. Resource Bricolage for Parallel Database Systems. *PVLDB* 8, 1 (2014), 25–36. http://www.vldb.org/pvldb/vol8/p25-Li.pdf

[19] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. 2018. Model-free Control for Distributed Stream Data Processing Using Deep Reinforcement Learning. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 705–718. https://doi.org/10.14778/3184470.3184474

[20] Jie Liu, Wenqian Dong, Dong Li, and Qingqing Zhou. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *Proc. VLDB Endow.* 14, 11 (2021), 1950–1963. http://www.vldb.org/pvldb/vol14/p1950-liu.pdf

[21] Yao Lu, Srikanth Kandula, Arnd Christian König, and Surajit Chaudhuri. 2021. Pre-training Summarization Models of Structured Datasets for Cardinality Estimation. *Proc. VLDB Endow.* 15, 3 (2021), 414–426. http://www.vldb.org/pvldb/vol15/p414-lu.pdf

[22] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, and Jingren Zhou. 2022. Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing. https://doi.org/10.48550/ARXIV.2207.02026

[23] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1248–1261. https://doi.org/10.1145/3448016.3457276

[24] Ryan Marcus and Olga Papaemmanouil. 2016. WiSeDB: A Learning-based Workload Management Advisor for Cloud Databases. *PVLDB* 9, 10 (2016), 780–791. http://www.vldb.org/pvldb/vol9/p780-marcus.pdf

[25] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[26] Ryan C. Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[27] Regina Marler and J S Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization* 26, 6 (2004), 369–395.

[28] MaxCompute [n.d.]. Open Data Processing Service. https://www.alibabacloud.com/product/maxcompute.

[29] Achille Messac. 2012. From Dubious Construction of Objective Functions to the Application of Physical Programming. *AIAA Journal* 38, 1 (2012), 155–163.

[30] Achille Messac, Amir Ismailyahaya, and Christopher A Mattson. 2003. The normalized normal constraint method for generating the Pareto frontier. *Structural and Multidisciplinary Optimization* 25, 2 (2003), 86–98.

[31] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[32] Parimarjan Negi, Ryan C. Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032. http://www.vldb.org/pvldb/vol14/p2019-negi.pdf

[33] Yuan Qiu, Yilei Wang, Ke Yi, Feifei Li, Bin Wu, and Chaoqun Zhan. 2021. Weighted Distinct Sampling: Cardinality Estimation for SPJ Queries. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1465–1477. https://doi.org/10.1145/3448016.3452821

[34] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. 2016. PerfOrator: eloquent performance models for Resource Optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*. 415–427. https://doi.org/10.1145/2987550.2987566

[35] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 99–113. https://doi.org/10.1145/3318464.3380584

[36] Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, and Prashant J. Shenoy. 2021. Spark-based Cloud Data Analytics using Multi-Objective Optimization. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 396–407. https://doi.org/10.1109/ICDE51399.2021.00041

[37] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319. https://doi.org/10.14778/3368289.3368296

[38] Ji Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation for Similarity Queries. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1745–1757. https://doi.org/10.1145/3448016.3452790

[39] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*. The Association for Computer Linguistics, 1556–1566. https://doi.org/10.3115/v1/p15-1150

[40] Zilong Tan and Shivnath Babu. 2016. Tempo: robust and self-tuning resource management in multi-tenant parallel databases. *Proceedings of the VLDB Endowment* 9, 10 (2016), 720–731.

[41] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB* 2, 2 (2009), 1626–1629.

[42] Immanuel Trummer and Christoph Koch. 2014. Approximation Schemes for Many-objective Query Optimization. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. ACM, New York, NY, USA, 1299–1310. https://doi.org/10.1145/2588555.2610527

[43] Immanuel Trummer and Christoph Koch. 2014. Multi-objective Parametric Query Optimization. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 221–232. https://doi.org/10.14778/2735508.2735512

[44] Immanuel Trummer and Christoph Koch. 2015. An Incremental Anytime Algorithm for Multi-Objective Query Optimization. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1941–1953. https://doi.org/10.1145/2723372.2746484

[45] Kapil Vaidya, Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. 2021. Leveraging Query Logs and Machine Learning for Parametric Query Optimization. *Proc. VLDB Endow.* 15, 3 (2021), 401–413. http://www.vldb.org/pvldb/vol15/p401-vaidya.pdf

[46] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[47] M. van Steen and A.S. Tanenbaum. 2017. *Distributed Systems* (3 ed.).

[48] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, Guy M. Lohman (Ed.). ACM, 5:1–5:16. https://doi.org/10.1145/2523616.2523633

[49] Lalitha Viswanathan, Alekh Jindal, and Konstantinos Karanasos. 2018. Query and Resource Optimization: Bridging the Gap. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1384–1387. https://doi.org/10.1109/ICDE.2018.00156

[50] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A Normalizing Flow based Cardinality Estimator. *Proc. VLDB Endow.* 15, 1 (2021), 72–84. http://www.vldb.org/pvldb/vol15/p72-li.pdf

[51] Lucas Woltmann, Dominik Olwig, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. PostCENN: PostgreSQL with Machine Learning Models for Cardinality Estimation. *Proc. VLDB Endow.* 14, 12 (2021), 2715–2718. http://www.vldb.org/pvldb/vol14/p2715-woltmann.pdf

[52] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *Proc. VLDB Endow.* 12, 3 (2018), 210–222. https://doi.org/10.14778/3291264.3291267

[53] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2009–2022. https://doi.org/10.1145/3448016.3452830

[54] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. 2020. BayesCard: Revitilizing Bayesian Frameworks for Cardinality Estimation. https://doi.org/10.48550/ARXIV.2012.14743

[55] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2463676.2465288

[56] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73. https://doi.org/10.14778/3421424.3421432

[57] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (2019), 279–292. https://doi.org/10.14778/3368289.3368294

[58] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J. Kim. 2019. Graph Transformer Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 11960–11970. https://proceedings.neurips.cc/paper/2019/hash/9d63484abb477c97640154d40595a3bb-Abstract.html

[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[60] Khaled Zaouk, Fei Song, Chenghao Lyu, Arnab Sinha, Yanlei Diao, and Prashant J. Shenoy. 2019. UDAO: A Next-Generation Unified Data Analytics Optimizer. *PVLDB* 12, 12 (2019), 1934–1937. https://doi.org/10.14778/3352063.3352103

[61] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 415–432. https://doi.org/10.1145/3299869.3300085

[62] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2102–2114. https://doi.org/10.1145/3448016.3457291

[63] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: a Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale. *Proc. VLDB Endow.* 7, 13 (2014), 1393–1404. https://doi.org/10.14778/2733004.2733012

[64] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *The VLDB Journal* 21, 5 (Oct. 2012), 611–636. https://doi.org/10.1007/s00778-012-0280-z

[65] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proc. VLDB Endow.* (2020), 1416–1428.

[66] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *Proc. VLDB Endow.* 14, 9 (2021), 1489–1502. http://www.vldb.org/pvldb/vol14/p1489-zhu.pdf

[67] Yiwen Zhu, Matteo Interlandi, Abhishek Roy, Krishnadhan Das, Hiren Patel, Malay Bag, Hitesh Sharma, and Alekh Jindal. 2021. Phoebe: A Learning-based Checkpoint Optimizer. *Proc. VLDB Endow.* 14, 11 (2021), 2505–2518. http://www.vldb.org/pvldb/vol14/p2505-zhu.pdf

[68] Yuqing Zhu and Jianxun Liu. 2019. ClassyTune: A Performance Auto-Tuner for Systems in the Cloud. *IEEE Transactions on Cloud Computing* (2019), 1–1.

[69] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: tapping the performance potential of systems via automatic configuration tuning. *SoCC '17: ACM Symposium on Cloud Computing Santa Clara California September, 2017* (2017), 338–350.