Multi-Objective Optimization for Data Analytics in the Cloud

Qi Fan

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2024IPPAX069

Thèse de doctorat

ÉCOLE POLYTECHNIQUE

IP PARIS

Multi-Objective Optimization for Data Analytics in the Cloud

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à École polytechnique

École doctorale n°626 Ecole Doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 23/09/2024, par

QI FAN

Composition du Jury :

Laurent d'Orazio Professor, CNRS IRISA, Rennes 1, University	Président & Rapporteur
Reza Akbarinia HDR, LIRMM, INRIA, University of Montpellier	Rapporteur
Peter J. Haas Professor, University of Massachusetts Amherst	Examinateur
Pierre Bourhis Junior Researcher, INRIA, CNRS, Universite Lille 1	Examinateur
Yanlei Diao Professor, École polytechnique	Directeur de thèse

Acknowledgements

I would like to take this opportunity to express my heartfelt gratitude to everyone who provided support and encouragement throughout my Ph.D. journey.

First and foremost, the completion of my dissertation would not have been possible without the support and nurturing of my supervisor, Prof. Yanlei Diao. Her dedicated mentorship and exceptional technical expertise have been invaluable to me. She has generously devoted her time to offer advice on reading, writing, presentations, and even technical communication skills. I am deeply grateful for her encouragement, patience, and continued support.

I would like to express my deepest appreciation to my committee members, Prof. Laurent D'Orazio, Dr. Reza Akbarinia, Prof. Peter J. Haas, and Dr. Pierre Bourhis, who engaged with my thesis diligently and enthusiastically, providing valuable suggestions. I am also extremely grateful to the jury members of the Comité de Suivi, Prof. Leo Liberti and Dr. Pierre Bourhis, for their guidance on this thesis.

I would like to extend my sincere thanks to my colleagues at École Polytechnique and Inria. I had a wonderful time working alongside my team members, Chenghao Lyu, Guillaume Lachaud, Vincent Jacob, Arnab Sinha, Fei Song, who offered invaluable help and advice, not only in our work but also in life. Special thanks to Ioana Manolescu and Oana Balalau; your help and valuable suggestions regarding research and my future career cannot be overestimated. I also greatly appreciate the time spent together with Oana Goga, Madhulika Mohanty and Garima Gaur, Abir Benzaamia, Asmaa El fraihi, Nardjes Amieur, Hiba, Ghufran Khan, Kun Zhang, Salim Chouaki, Théo Bouganim, Théo Galizzi, Tom Calamai, Simon Ebel, Nelly Barret, Khaled Zaouk, Luciano di Palma, and Pawel Guzewicz. You all contribute to a pleasant work environment, where we enjoy fascinating experiences of sharing

food from different countries and engaging in interesting discussions on various cultures and religions. In addition, I'd like to acknowledge the assistance of Frédéric Ayrault, Jessica Gameiro, Hélèna Kutniak and Bahareh Yazdi with administrative matters.

I would like to thank my friends Enhui Huang, Guili Zhao and Mengyu Gao for their companionship in France. It has been an unforgettable experience, especially living in a foreign country.

I would like to thank the China Scholarship Council (CSC) and European Research Council (ERC) for their funding of my Ph.D. studies in France. This experience and the memories will remain in my heart forever.

Finally, I would like to express my deepest gratitude to my parents, my sister, and my boyfriend for their endless love, understanding, and spiritual support.

Thank you all!
Merci à tous!
Qi FAN

Abstract

Big data query processing has become increasingly important, prompting the development and cloud deployment of numerous systems. However, automatically tuning the numerous parameters in these big data systems introduces growing complexity in meeting users' performance goals and budgetary constraints. Determining optimal configurations is challenging due to the need to address: 1) multiple competing performance goals and budgetary constraints, such as low latency and low cost, 2) a high-dimensional parameter space with complex parameter control, and 3) the requirement for high computational efficiency in cloud use, typically within 1-2 seconds.

To address the above challenges, this thesis proposes efficient multi-objective optimization (MOO) algorithms for a cloud optimizer to meet various user objectives. It computes Pareto optimal configurations for big data queries within a high-dimensional parameter space while adhering to stringent solving time requirements. More specifically, this thesis introduces the following contributions.

The first contribution of this thesis is a benchmarking analysis of existing MOO methods and solvers, identifying their limitations, particularly in terms of efficiency and the quality of Pareto solutions, when applied to cloud optimization.

The second contribution introduces MOO algorithms designed to compute Pareto optimal solutions for query stages, which are units defined by shuffle boundaries. In production-scale big data processing, each stage operates within a high-dimensional parameter space, with thousands of parallel instances. Each instance requires resource parameters determined upon assignment to one of thousands of machines, as exemplified by systems like MaxCompute. To achieve Pareto optimality for each query stage, we propose a novel hierarchical MOO approach. This method de-

composes the stage-level MOO problem into multiple parallel instance-level MOO problems and efficiently derives stage-level MOO solutions from instance-level MOO solutions. Evaluation results using production workloads demonstrate that our hierarchical MOO approach outperforms existing MOO methods by 4% to 77% in terms of latency and up to 48% in cost reduction while operating within 0.02 to 0.24 seconds compared to current optimizers and schedulers.

Our third contribution aims to achieve Pareto optimality for the entire query with finer-granularity control of parameters. In big data systems like Spark, some parameters can be tuned independently for each query stage, while others are shared across all stages, introducing a high-dimensional parameter space and complex constraints. To address this challenge, we propose a new approach called Hierarchical MOO with Constraints (HMOOC). This method decomposes the optimization problem of a large parameter space into smaller subproblems, each constrained to use the same shared parameters. Given that these subproblems are not independent, we develop techniques to generate a sufficiently large set of candidate solutions and efficiently aggregate them to form global Pareto optimal solutions. Evaluation results using TPC-H and TPC-DS benchmarks demonstrate that HMOOC outperforms existing MOO methods, achieving a 4.7% to 54.1% improvement in hypervolume and an 81% to 98.3% reduction in solving time.

Resumé

Le traitement des requêtes Big Data est devenu de plus en plus important, ce qui a conduit au développement et au déploiement dans le cloud de nombreux systèmes. Cependant, le réglage automatique des nombreux paramètres de ces systèmes Big Data introduit une complexité croissante pour répondre aux objectifs de performance et aux contraintes budgétaires des utilisateurs. La détermination des configurations optimales est un défi en raison de la nécessité de prendre en compte : 1) plusieurs objectifs de performances et contraintes budgétaires concurrents, tels qu'une faible latence et un faible coût, 2) un espace de paramètres de grande dimension avec un contrôle de paramètres complexe, et 3) l'exigence d'une configuration élevée. efficacité de calcul dans l'utilisation du cloud, généralement en 1 à 2 secondes.

Pour relever les défis ci-dessus, cette thèse propose des algorithmes d'optimisation multi-objectifs (MOO) efficaces pour un optimiseur de cloud afin de répondre à divers objectifs des utilisateurs. Il calcule les configurations Pareto optimales pour les requêtes Big Data dans un espace de paramètres de grande dimension tout en respectant des exigences strictes en matière de temps de résolution. Plus précisément, cette thèse présente les contributions suivantes.

La première contribution de cette thèse est une analyse comparative des méthodes et solveurs MOO existants, identifiant leurs limites, notamment en termes d'efficacité et de qualité des solutions Pareto, lorsqu'elles sont appliquées à l'optimisation du cloud.

La deuxième contribution présente des algorithmes MOO conçus pour calculer des solutions optimales de Pareto pour les étapes de requête, qui sont des unités définies par des limites de mélange. Dans le traitement de données volumineuses à l'échelle de la production, chaque étape fonctionne dans un espace de paramètres de

grande dimension, avec des milliers d'instances parallèles. Chaque instance nécessite des paramètres de ressources déterminés lors de l'affectation à l'une des milliers de machines, comme l'illustrent des systèmes comme MaxCompute. Pour atteindre l'optimalité de Pareto pour chaque étape de requête, nous proposons une nouvelle approche MOO hiérarchique. Cette méthode décompose le problème MOO au niveau de l'étape en plusieurs problèmes MOO parallèles au niveau de l'instance et dérive efficacement des solutions MOO au niveau de l'étape à partir de solutions MOO au niveau de l'instance. Les résultats de l'évaluation utilisant des charges de travail de production démontrent que notre approche MOO hiérarchique surpasse les méthodes MOO existantes de 4 à 77% en termes de latence et jusqu'à 48% en réduction des coûts tout en fonctionnant dans un délai de 0,02 à 0,24 seconde par rapport aux optimiseurs et planificateurs actuels.

Notre troisième contribution vise à atteindre l'optimalité Pareto pour l'ensemble de la requête avec un contrôle plus fin des paramètres. Dans les systèmes Big Data comme Spark, certains paramètres peuvent être ajustés indépendamment pour chaque étape de la requête, tandis que d'autres sont partagés entre toutes les étapes, introduisant ainsi un espace de paramètres de grande dimension et des contraintes complexes. Pour relever ce défi, nous proposons une nouvelle approche appelée MOO hiérarchique avec contraintes (HMOOC). Cette méthode décompose le problème d'optimisation d'un grand espace de paramètres en sous-problèmes plus petits, chacun contraint d'utiliser les mêmes paramètres partagés. Étant donné que ces sous-problèmes ne sont pas indépendants, nous développons des techniques pour générer un ensemble suffisamment large de solutions candidates et les agréger efficacement pour former des solutions Pareto optimales globales. Les résultats de l'évaluation utilisant les benchmarks TPC-H et TPC-DS démontrent que HMOOC surpasse les méthodes MOO existantes, obtenant une amélioration de 4,7% à 54,1% de l'hypervolume et une réduction de 81% à 98,3% du temps de résolution.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The prevalence of cloud technology among big data users is notable both currently and is expected to increase in the foreseeable future. Presently, approximately 94% of global companies utilize cloud software [91]. Gartner projects that by 2025, over 95% of new digital workloads will transition to cloud-native platforms, a significant increase from 30% in 2021 [28]. Furthermore, it is anticipated that by 2028, 70% of all workloads will be conducted within cloud computing environments [27].

Running cloud-based analytical tasks presents significant challenges for users, particularly due to the multitude of requirements involved. Consider an enterprise user intending to execute a Spark analytic task [109] in the cloud. Initially, the user must select an appropriate hardware configuration from over 300 available instances on Amazon EC2 [2], each offering various combinations of CPU power, memory, storage, and networking capabilities. Subsequently, the user is required to configure several software parameters, such as the number of CPU cores, memory size, number of executors, shuffle behaviors (e.g., number of shuffle partitions), and memory management (e.g., fraction of memory allocated for execution and storage). These configurations critically influence both performance metrics, such as latency, and monetary costs, in accordance with the pricing strategy of Amazon EC2.

The necessity for automatic optimization of user analytic tasks stems from the need to fulfill diverse user requirements, encompassing factors like latency, monetary cost, and additional objectives. Typically, cloud service providers mandate users to choose a cloud instance—a set of machines tailored with specific computing capa-

bilities—suitable for their application demands. Nonetheless, the manual scaling, deployment, and configuration of cloud services often result in suboptimal settings that diminish performance [6], thereby escalating latency and monetary costs.

The predominant theoretical approach for optimizing multiple, potentially conflicting, user performance objectives is Multi-Objective Optimization (MOO). MOO identifies a Pareto-optimal set of solutions that illustrates the trade-offs among various objectives, with no single solution outperforming others across all dimensions. This framework empowers users to select the optimal solution according to their preferences. For example, in balancing two competing objectives—latency and cost—the solution set might be categorized into three distinct groups: low latency at a higher cost, medium latency at a moderate cost, and high latency at a reduced cost. Consequently, MOO provides the flexibility for users to choose a solution tailored to specific preferences, such as low latency.

The Pareto optimal solution yielded by MOO is identified as a *configuration* after an extensive search through a vast parameter space in big data systems. A *configuration* for cloud analytical tasks is defined as a constant vector that assigns specific values to each tuning parameter within a fixed and ordered set. For instance, the configuration [1,5] for the parameters [CPU cores,memory size] establishes a system setting with 1 CPU core and 5 GB of memory. This *configuration* is utilized by cloud analytical tasks to operate within big data systems, aiming to achieve performance goals that encompass multiple objectives, such as reducing latency and minimizing monetary costs.

In this thesis, we propose the development of a *Cloud Optimizer* grounded in MOO principles. The primary aim of this optimizer is to *automatically determine the optimal configuration for each incoming analytical task across a range of objectives, optimizing them simultaneously.* The *Cloud Optimizer* is designed to accommodate both SQL and Machine Learning (ML) tasks. At its core, it employs a systematic MOO methodology to generate a set of optimal configurations, from which the most suitable configuration is selected to best satisfy all specified objectives.

1.1 Technical Challenges

The primary distinction between our research and existing studies is our approach to addressing the MOO problem with detailed, complex parameter control under strict time constraints. While state-of-the-art methods [35, 49, 55, 96] tackle MOO in parameter tuning for big data systems, they do not typically differentiate between types of parameters or accommodate the stringent 1-2 second time limits that are characteristic of cloud environments. In contrast, a recent study, UDO [102], classifies database system parameters into heavy and light categories based on the cost of value modification. Nevertheless, UDO's focus is on optimizing throughput or latency exclusively, rather than addressing broader MOO challenges.

In particular, we tackle the following technical challenges.

- **Good quality and coverage of the cloud optimizer:** Fundamentally distinct from single-objective optimization, MOO introduces significant challenges for a *Cloud Optimizer* tasked with exploring the *Pareto frontier*—the set of all optimal solutions in MOO. Achieving *good quality and coverage* along this frontier means identifying solutions that not only perform optimally across all objectives but also represent a diverse range of preferences within the objective space. Traditional MOO methods such as the Weighted Sum approach [19] often struggle to provide comprehensive coverage of the Pareto frontier. On the other hand, evolutionary algorithms like NSGA-II [22] promote diversity but typically offer only an approximation of the Pareto frontier. Moreover, the quality of solutions and the computation time are heavily influenced by hyperparameters such as population size and the number of evaluations.

- **High efficiency of the cloud optimizer:** Achieving *high efficiency* presents a considerable challenge, as the *Cloud Optimizer* must recommend configurations within stringent time constraints typical for cloud usage, such as the 1-2 second window necessary to avoid delays in starting cloud-based analytical tasks. Previous research on MOO for Spark tuning [86] demonstrated that the Multi-Objective Bayesian Optimization method [20] typically requires about

48 seconds to deliver *Pareto solutions*. Such extended processing times are highly impractical for cloud environments, underscoring the need for more rapid solution methodologies.

- **High-dimensional parameters:** In practical applications, the number of tuning parameters can be vast due to unique workload characteristics. For instance, service providers like Alibaba Cloud are tasked with assigning *resource plans* to all instances within a query stage, defined by shuffle boundaries. These *resource plans* dictate the allocation of resources, such as CPU cores and memory size, to each instance on a specific machine [64, 71]. In scenarios where a stage includes m instances, the parameter space expands dramatically to $2 \times m$, with m potentially surpassing 6,000 in real workloads. The use of the Evolutionary method [22], although powerful, proves impractical in such contexts as it may require more than 10 minutes to deliver Pareto solutions for stages comprising thousands of instances, rendering it unsuitable for the rapid-response demands of Alibaba Cloud's operational environment.

- **Fine-grained complex control of parameters:** Unlike previous works that uniformly tune parameters across queries [12, 26, 108, 112, 114, 119], cloud engines like Spark [109] offer a diverse range of parameters, which can be categorized into three groups. *Context parameters* ($\boldsymbol{\theta_c} \subseteq \mathbb{R}^{d_c}$) are used to initialize the Spark context, allocating shared resources and influencing runtime behaviors such as shuffling. *Query plan parameters* ($\boldsymbol{\theta_p} \subseteq \mathbb{R}^{d_p}$) govern the translation from logical to physical query plans. *Query stage parameters* ($\boldsymbol{\theta_s} \subseteq \mathbb{R}^{d_s}$) optimize individual query stages within the physical plan. The parameters $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ offer fine-grained control over each query stage, creating a high-dimensional parameter space. For example, if a query comprises m stages, the total number of fine-grained parameters is calculated as $d_c + (d_p + d_s) \times m$, leading to a significantly large parameter space when m is substantial.

  These diverse parameters introduce significant complexity in controlling different stages of a query. For example, in Spark [109], $\boldsymbol{\theta_c}$ manages shared resources across all stages and must be established at query submission to

initialize the Spark context. In contrast, $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ are ideally fine-tuned per stage at runtime, taking advantage of precise, real-time statistics. This intricate control scheme presents substantial challenges, necessitating a hybrid approach that merges compile-time and runtime optimization techniques. Additionally, the interdependencies among these parameters further complicate the optimization process.

1.2 Contributions

To address the challenge of ensuring both efficiency and Pareto optimality in MOO problems involving high-dimensional parameters and complex controls within cloud optimization contexts, we propose novel MOO approaches. These are designed to enhance the performance and effectiveness of cloud optimizers. Our contributions are outlined as follows:

1.2.1 Benchmark study of Multi-Objective Optimization

A benchmark study is conducted to evaluate various MOO algorithms and existing solvers. Approaches to addressing MOO generally categorize into two distinct methodologies: transforming MOO problems into single-objective optimization problems or solving the MOO problems directly. While optimizing a single objective involves the use of specific solvers, addressing MOO directly has attracted considerable research interest, as evidenced by significant contributions in the literatures [20, 22, 69]. To determine the efficacy of existing solvers and MOO algorithms within the context of the *Cloud Optimizer*, we analyze the performance of two commercial solvers alongside several state-of-the-art MOO methods.

Study on Solvers. (Section 4.1) Our optimization challenge is categorized as a Mixed Integer Non-Linear Programming (MINLP) problem, stemming from the presence of non-linear objective functions and a diverse range of parameter types, which include both continuous and integer variables. These non-linear objectives originate from learned models employed to avoid the resource-intensive evaluation of operational configurations in large-scale data systems. An example of such non-

linear behavior can be observed in the Rectified Linear Unit (ReLU), a commonly used activation function in Deep Neural Networks (DNNs).

Taking DNNs equipped with ReLU activation functions as a case study for our objective function, we analyze the performance of the commercial solver: Knitro [42]. Knitro, a Non-Linear Programming (NLP) solver, is adept at directly handling MINLP problems that incorporate DNN objectives.

Experimental Evaluation of Solvers and MOO Algorithms. (Section 4.2) We employ various performance metrics to evaluate solvers and MOO algorithms. For solvers tackling single-objective optimization problems, we assess solution quality based on the objective values, while efficiency is measured by the time taken to solve the problem. In contrast, MOO algorithms produce a set of solutions that highlight trade-offs among diverse user preferences. To evaluate these, we use hypervolume (HV) [41] [4], which quantifies the extent of the space dominated by the Pareto solutions, alongside solving time to gauge both the performance and efficiency of MOO methods.

In our experimental evaluation, we compare the performance of three solvers: Knitro, grid search, and random sampling. While Knitro yields objective results comparable to those of grid search and random sampling, it requires significantly more solving time, exceeding 40 minutes.

We implemented and evaluated existing MOO methods, assessing both the HV of the solutions returned and the time required for solving. The methods tested include the Weighted Sum (WS) approach [69], the Evolutionary (EVO) method [22], and Multi-Objective Bayesian Optimization (MOBO) [20], each representing a distinct category of MOO strategies. The WS method transforms an MOO problem into a single-objective optimization challenge by applying weights to each objective, facilitating the use of a single-objective solver. The EVO method, a randomized approach, employs genetic operations such as crossover and mutation on randomly initialized populations. Conversely, MOBO extends Bayesian Optimization (BO) to MOO problems by modeling each objective function with a surrogate model and designing an acquisition function to identify new points likely to yield Pareto optimal solutions. In comparative analysis, both WS and EVO methods outperformed

MOBO in terms of HV, achieving more than 72.2% compared to 41.8%, while also requiring significantly less solving time—less than 0.03 seconds compared to MOBO's 290 seconds.

Given the stringent demands for high quality and efficiency imposed by the *Cloud Optimizer*, we have chosen grid search and random sampling as the primary solvers. Additionally, the WS and EVO methods are selected as the baseline MOO approaches for subsequent experiments.

1.2.2 Stage-level Optimizer

We propose a novel MOO algorithm specifically designed to compute Pareto optimal solutions for *stages* which are distinct units defined by shuffle boundaries in large-scale big data processing. Within these environments, such as those managed by Alibaba Cloud's MaxCompute system [71], each stage functions within a high-dimensional parameter space, handling thousands of parallel instances. Each instance requires specific resource allocations across a multitude of machines. This scenario constitutes a Multi-Objective Resource Optimization (MORO) problem, where multiple performance goals, including stage latency and cost, are considered simultaneously. Aimed at minimizing these stage-level objectives concurrently, our approach seeks to achieve Pareto optimality in MORO while enhancing efficiency for cloud usage. The main contributions of this approach are detailed in Chapter 5.

System design and our approach. (Section 5.1 and 5.2) We augmented the MaxCompute system with a resource optimizer tailored for each stage. The core design principle involves segmenting the MORO problem into a series of more manageable sub-problems. Each sub-problem is strategically tackled by two specialized modules: the Intelligent Placement Advisor (IPA) and the Resource Assignment Advisor (RAA). The IPA module is engineered to allocate instances to specific machines effectively, whereas the RAA module is focused on fine-tuning the resource allocations for each instance to optimize the balance between stage latency and cost.

To address the MORO problem under stringent time constraints, we developed a two-step approach, each corresponding to one of the extended modules. In Step 1, the IPA aims to minimize stage latency by optimally allocating instances to

machines, considering factors such as machine capacities and individual instance latencies. Step 2 involves the RAA, which refines the resources assigned to each instance on a designated machine to achieve optimal outcomes in stage latency, cost, and other relevant objectives. The detailed formulations of this two-step approach are explored in Section 5.2. This thesis primarily concentrates on the MOO strategies employed in Step 2 (RAA), which involves intricate fine-tuning of resource parameters across multiple objectives.

Resource Assignment Advisor. (Section 5.3) We introduce a hierarchical MOO approach designed to fine-tune the resources allocated to each instance, with the aim of reducing stage latency, cost, and other objectives. This method effectively decomposes the complex stage-level MOO problem into multiple, parallel instance-level MOO problems. From these, it efficiently synthesizes stage-level MOO solutions that are derived from the solutions at the instance level. Furthermore, our approach is proven to achieve Pareto optimality at the stage level.

Experimental evaluation. (Section 5.4) Utilizing production workloads comprising 0.6 million jobs and 1.9 million stages, alongside a simulator of the enhanced MaxCompute environment, our evaluation yields promising results: 1) Relative to the default MaxCompute settings, our approach significantly improves performance, achieving latency reductions between 36% and 80% and cost savings from 29% to 75%, all while maintaining solving times between 0.02 to 0.24 seconds. 2) When compared to existing MOO methods, our approach consistently delivers solutions for all workloads within 0.24 seconds. In contrast, existing MOO methods often fail to generate feasible solutions for some queries within a generous 60-second timeframe. Furthermore, our method achieves a reduction in stage latency by 4-77% and a decrease in stage cost by up to 48% relative to these methods. 3) Moreover, our approach demonstrates adaptive resource allocation capabilities, assigning increased resources to longer-running jobs in comparison to MaxCompute's default settings.

1.2.3 Query-level Optimizer

We have developed a novel MOO algorithm aimed at achieving Pareto optimality across entire queries, facilitated by finer-granularity control over parameters. In

complex big data systems like Spark [109], parameter tuning is intricate due to the parameter settings: some parameters can be independently adjusted for each query stage, while others are global, affecting all stages and thus creating a high-dimensional parameter space with complex constraints. Specifically, global parameters are set at compile time, dictating behaviors such as resource allocation during query execution, and cannot be modified during runtime, unlike other parameters that are adjustable during runtime. Our primary contributions, designed to address these challenges, are outlined as follows:

Hybrid compile-time/runtime optimization overview. (Section 6.1) We introduce a hybrid optimization strategy that combines compile-time and runtime approaches to effectively manage the diversity and complex control of parameters in Spark. At compile time, optimal configurations for shared parameters are determined using estimated statistics, with a careful consideration of their interdependencies with other parameters. Subsequently, during runtime, adjustments to other parameters are made for each query stage, based on real-time statistics and the specific structure of the query. This thesis specifically emphasizes the MOO strategies applied during the compile-time phase, focusing on addressing the challenges of parameter diversity and achieving fine-grained control.

Compile-time optimization. (Section 6.2 and 6.3) We propose a novel approach, termed Hierarchical MOO with Constraints (HMOOC), aimed at reducing query-level latency and cost. This method systematically decomposes the overarching optimization problem within a large parameter space into smaller, more manageable subproblems. Each subproblem is bound by the requirement to utilize shared parameters, acknowledging their interdependence. To address these interlinked challenges, we develop techniques to generate a comprehensive set of candidate solutions. Additionally, we introduce various aggregation methods to efficiently synthesize these candidates into global Pareto optimal solutions. A detailed theoretical analysis of our proposed approach, including proofs and a complexity analysis, is presented in Section 6.3.

Experimental evaluation (Section 6.4) We conducted optimization analyses using datasets from the TPC-H and TPC-DS benchmarks, focusing on the following

evaluations: 1) We assessed the performance of three proposed aggregation methods, all of which are designed to ensure the return of at least a subset of query-level optimal solutions, based on sub-query level optimal inputs. The most effective method achieved the optimal configuration in only 0.5-0.8 seconds for all TPC-H and TPC-DS queries. 2) When compared to existing MOO methods, our compile-time MOO algorithm (HMOOC), tailored for fine-grained parameter tuning, demonstrated improvements in hypervolume (measuring the space dominated by the Pareto front) by 4.7%-54.1% and achieved reductions in solving time by 81%-98.3%. 3) In comparison to query-level tuning, which applies uniform configurations across all sub-queries without finer-grained adjustments, our approach significantly outperformed in both hypervolume (improving from 87% to 93%) and solving time (reducing from an average of over 14 seconds to just 0.8 seconds).

1.3 Thesis Outline

The thesis is organized as follows. Chapter 2 introduces state-of-the-art MOO approaches and related works on parameter tuning and resource optimization in big data systems. Chapter 3 provides a general definition of the MOO problem in big data systems and an overview of the corresponding system. In Chapter 4, we present a benchmark study of MOO, including existing solvers and MOO algorithms. Chapters 5 and 6 develop MOO algorithms for the *cloud optimizer* in two popular big data systems: MaxCompute [71] and Spark [109]. Finally, Chapter 7 summarizes the thesis and discusses future work.

CHAPTER 2

Basics and Related Work

This chapter introduces the Multi-Objective Optimization (MOO) problem by surveying existing MOO algorithms and discussing various performance indicators. Additionally, it reviews state-of-the-art research related to parameter tuning and resource optimization in cloud-based data analytics.

2.1 Multi-Objective Optimization (MOO)

2.1.1 Definitions

The general multi-objective optimization (MOO) problem could be defined as follows:

Definition 2.1.1. General Multi-Objective Optimization (MOO) problem.

$$\begin{aligned} \arg\min_{\boldsymbol{x}} \quad & F(\boldsymbol{x}) = \quad [F_1(\boldsymbol{x}), ..., F_k(\boldsymbol{x})] \\ & g_l(\boldsymbol{x}) \leq 0, \quad l \in 1, 2, ..., m \\ s.t. \quad & h_e(\boldsymbol{x}) = 0, \quad e \in 1, 2, ..., n \\ & \boldsymbol{x} \in \mathbb{R}^d \end{aligned} \tag{2.1}$$

where k is the number of objectives, m is the number of inequality constraints, and n is the number of equality constraints. $\boldsymbol{x}$ is the *decision variable* with d dimensions.

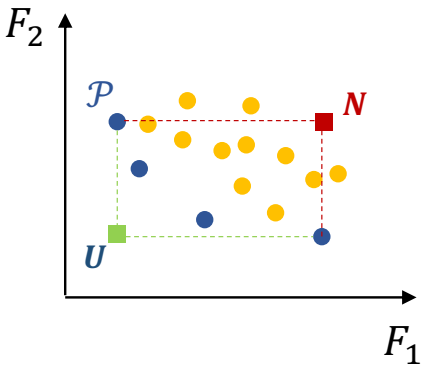


Figure 2.1: Example of MOO definitions

Usually, the objectives in MOO compete with each other, leading to trade-offs among different objectives. Consequently, unlike single-objective optimization, which yields a single optimal solution, MOO returns a set of solutions.

Definition 2.1.2. Pareto domination. A point $F(\boldsymbol{x}^*)$ *Pareto-dominates* another point $F(\boldsymbol{x})$ iff $\forall i \in [1,k], F_i(\boldsymbol{x}^*) \leq F_i(\boldsymbol{x})$ and $\exists j \in [1,k], F_j(\boldsymbol{x}^*) < F_j(\boldsymbol{x})$, where $\boldsymbol{x}, \boldsymbol{x}^* \in \mathbb{R}^d$.

Definition 2.1.3. Pareto optimal. A point $F(\boldsymbol{x}^*)$ is *Pareto optimal* iff there does not exist another point that Pareto-dominates it.

Definition 2.1.4. Pareto set (frontier). The *Pareto set (frontier)* $\mathcal{P}$ includes all the Pareto optimal solutions in the objective space and it is the solution to the MOO problem.

Definition 2.1.5. Utopia point. A point $\boldsymbol{U}$ is *Utopia* iff for each $i = 1, 2, \ldots, k$, $U_i = \underset{\boldsymbol{x}}{\text{minimum}}\{F_i(\boldsymbol{x}) | \boldsymbol{x} \in \mathbb{R}^d, F_i(\boldsymbol{x}) \in \mathcal{P}\}$.

Definition 2.1.6. Nadir point. A point $\boldsymbol{N}$ is *Nadir* iff for each $i = 1, 2, \ldots, k$, $N_i = \underset{\boldsymbol{x}}{\text{maximum}}\{F_i(\boldsymbol{x}) | \boldsymbol{x} \in \mathbb{R}^d, F_i(\boldsymbol{x}) \in \mathcal{P}\}$.

Utopia and *Nadir* points do not exist in the objective space; rather, they are virtual constructs used to bound the objective space formed by all the Pareto solutions $\mathcal{P}$.

Figure 2.1 illustrates the aforementioned definitions with two competing objectives, F_1 and F_2. All circle points represent solutions in the objective space, where the yellow points are dominated by the blue points. The blue points are *Pareto*

optimal and form the *Pareto frontier* $\mathcal{P}$, as they cannot be dominated by any other points. The *Utopia* (green squared point) and *Nadir* (red squared point) points, $\boldsymbol{U}$ and $\boldsymbol{N}$ respectively, are derived from the *Pareto frontier* (blue points), representing the minimum and maximum values of each objective.

2.1.2 MOO Algorithms

This section reviews state-of-the-art works on MOO algorithms, categorized into preference-based algorithms, evolutionary algorithms, Multi-Objective Bayesian Optimization (MOBO) algorithms, and other existing MOO algorithms.

Preference-based Algorithms. Preference-based algorithms convert a MOO problem into a single-objective optimization by specifying preferences for different objectives. These preferences can be determined based on user preference or the goal of covering the entire Pareto front.

The Weighted Sum (WS) approach transforms a MOO problem into a single-objective optimization problem by assigning weights to different objective functions. While it is easy to implement, it often fails to provide good coverage over the Pareto front [19, 69].

The ϵ-constraint method [31] transforms a MOO problem into a constrained single-objective optimization problem by optimizing a single objective (e.g., the i-th objective) while treating the other objectives as constraints with user-defined values $\epsilon_j, j = 1, \ldots, k, j \neq i$. However, the optimality of this method depends on the choices of these user-defined values, making the hyperparameters difficult to determine.

Similarly, the Normalized Normal Constraint (NNC) method [74] transforms a MOO problem into a series of constrained single-objective optimization problems by forming a set of evenly distributed index points. A post-processing filter is then required to eliminate non-Pareto solutions. However, the number of Pareto solutions is limited to the number of index points, and the method can face time efficiency issues.

Overall, the preference-based MOO algorithms discussed above transform the MOO problem into either unconstrained or constrained single-objective optimiza-

tion problems. However, their performance may be affected by issues such as low efficiency, poor coverage, or suboptimal hyperparameter selection.

Evolutionary Algorithms. Evolutionary algorithms are population-based methods inspired by natural evolution. Starting with a randomly initialized population of solutions, these algorithms iteratively refine potential solutions using genetic operations such as selection, crossover, and mutation. They can be highly efficient since they do not require gradient information.

A popular multi-objective evolutionary algorithm, the Nondominated Sorting Genetic Algorithm-II (NSGA-II) [22], aims to generate a set of Pareto solutions by using dominance rank while maintaining diversity through crowding distance among individuals. It is a randomized method designed to return an approximation of the Pareto set. NSGA-III [21] improves upon NSGA-II by enhancing diversity maintenance through the use of well-distributed reference points on a hyperplane, replacing the crowding distance operator. The offspring population is then selected based on proximity to these reference points.

MOEA/D [111] decomposes a general MOO problem into multiple scalar optimization subproblems, each associated with a uniform-spread weight vector to convert the approximation of the Pareto frontier into a single-objective optimization problem, such as through the Weighted Sum approach. These subproblems can be optimized in parallel. Leveraging the idea that information from neighboring weight vectors can aid in updating neighboring solutions, MOEA/D reduces the complexity of generating new solutions by utilizing information from these adjacent weight vectors.

SPEA2 [120] addresses the MOO problem by evaluating individuals based on domination and maintaining diversity through the density of individuals. It assigns fitness values to individuals using a domination count, where a count of 0 indicates that an individual is non-dominated by others.

Although evolutionary algorithms can quickly provide results, they only yield an approximation of the Pareto frontier and their performance is sensitive to hyperparameters related to genetic operations, such as the population size and the number of iterations.

Multi-Objective Bayesian Optimization (MOBO) Algorithms. Bayesian Optimization (BO) can be extended to solve MOO problems. Traditional BO optimizes expensive-to-evaluate black-box functions by using a surrogate model to approximate the black-box function and an acquisition function to determine the next evaluation point [104]. In the context of MOO, each objective function is modeled with its own surrogate model, and the acquisition function is designed to search for new points that are likely to be Pareto optimal. Sequential exploration and batch exploration are two methods for exploring new points.

Sequential exploration. In MOBO with sequential selection, ParEGO [47] addresses MOO by transforming multiple objectives into a single objective in each iteration. It then maximizes the expected improvement (EI) using an evolutionary algorithm. PESMO [36] selects the next point by maximizing the expected reduction in the entropy of the posterior distribution over the Pareto set in each iteration. The Pareto set sample is obtained using grid search or NSGA-II. However, optimizing the acquisition function is computationally expensive, with a complexity that scales cubically with the sample size of the Pareto set, and the performance that depends on the number of Monte Carlo samples used. The UseMO [9] framework is designed to approximate the true Pareto frontier of MOO problems with black-box objectives using MOBO. It solves the MOO problem with multiple acquisition functions, selecting new points based on the maximum volume of the uncertainty hyperrectangle formed by the lower and upper confidence bounds of the surrogate models for multiple objectives. However, it primarily integrates the existing MOO solver, NSGA-II, into the BO process to determine the next point to explore.

Batch exploration. Parallel Expected Hypervolume Improvement (qEHVI) [20] designs its acquisition function to maximize the hypervolume improvement (HVI) between one or multiple candidate points and a set of reference points, which serve a similar role to *Nadir points*. The newly explored points are considered potential Pareto optimal solutions. Although qEHVI searches multiple points simultaneously, it suffers from high computational complexity, with the HVI calculation being exponential in the number of candidates. Diversity-Guided Efficient Multi-Objective Optimization (DGEMO) [48] explores multiple points while considering diversity.

It divides the performance space into sub-regions and, within each previously unexplored sub-region, greedily selects one point that provides the maximum hypervolume improvement with respect to the global Pareto frontier. Lin et al. [57] proposes a method for learning the Pareto set in MOBO that incorporates preferences. In each iteration, the method selects a small number of points to maximize the hypervolume improvement relative to the points already evaluated. However, the optimality of this approach may be compromised by the potential limitations of the learned model's performance.

A major limitation of the BO-based approach is the extended time required to produce a satisfactory Pareto set. This inefficiency poses challenges in meeting the stringent time constraints of cloud optimization.

Other state-of-the-art works. LaMOO [116] is a multi-objective optimizer that demonstrates superior sample efficiency empirically. The approach consists of two main steps: first, it learns a model from the observed samples to partition the search space. Then, for the promising regions likely containing a subset of Pareto solutions, it applies state-of-the-art MOO methods for exploration. While LaMOO measures efficiency based on the number of samples rather than exact time cost, it presents empirical results without providing a theoretical analysis of optimality for general MOO problems.

The Progressive Frontier (PF) approach [86] aims to incrementally transform the MOO problem into a series of constrained optimization (CO) problems (i.e., single-objective optimization problems), each of which can be solved individually. It computes a Pareto optimal set of configurations to reveal trade-offs among different objectives, recommends new configurations that best explore these trade-offs within a few seconds. However, its efficiency is limited in high-dimensional parameter spaces.

A cloud optimizer based on MOO demands both optimality and high efficiency for cloud use. In real scenarios, where hundreds or thousands of parameters need to be optimized, state-of-the-art MOO methods suffer from low efficiency.

2.1.3 Performance Indicators

Unlike single-objective optimization, MOO returns a set of solutions called the *Pareto frontier*. Visualization tools facilitate the comparison of different Pareto fronts in bi-objective problems. However, for optimization problems with more than two objectives, quantification of the Pareto front is more suitable, as visualizing higher-dimensional Pareto fronts is challenging [41].

The performance metrics of a *Pareto frontier* can be divided into four categories [41] [4]:

1. *Cardinality*: the number of non-dominated solutions generated by a MOO algorithm.
2. *Convergence*: the approaching to the true Pareto frontier.
3. *Uniformity*: the evenness of a Pareto frontier approximation.
4. *Distribution*: the distribution and coverage of a Pareto frontier approximation.

Different performance metrics exhibit various properties. For *Cardinality*, intuitively, a higher cardinality indicates better performance. However, it fails to represent diverse user preferences, particularly when non-dominated solutions are clustered in a small objective region. Additionally, it does not convey the quality of Pareto solutions when compared to other Pareto frontiers. For *Convergence*, most metrics require knowledge of the true Pareto front for evaluation. However, in practice, evaluating the true Pareto frontier is often not feasible. *Uniformity* measures the diversity of a Pareto frontier, with evenly distributed Pareto solutions indicating the capability to meet various user preferences. However, it does not capture how well a Pareto frontier performs in the objective space compared to another Pareto frontier. *Distribution* indicates how Pareto solutions are spread in the objective space, capturing dominance information when compared to other Pareto frontiers.

Our optimization problem requires a performance indicator that includes properties of both *Uniformity* and *Distribution* for the following reasons. First, we do not have knowledge of the true Pareto frontier. Additionally, when comparing different MOO baselines, it is essential to ensure both the quality and coverage of the

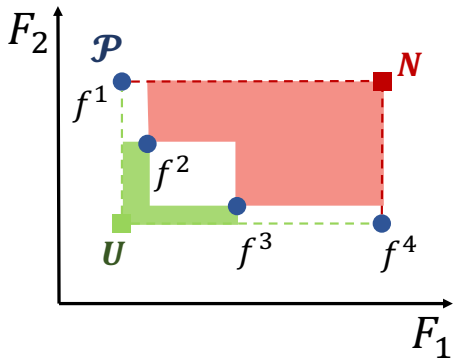


Figure 2.2: Examples of performance indicators (The hypervolume is highlighted in red, while the uncertainty space corresponds to the white region enclosed by the four blue points.)

Pareto solutions. This means that no other solutions should perform better in all objectives, and the solutions should cover a diverse range of the objective space to meet various user preferences.

Two example performance indicators suitable for our MOO problem are as follows.

Hypervolume Indicator. The Hypervolume Indicator (HV) [41] [4] measures the volume of the objective space dominated by the Pareto front. As illustrated in Figure 2.2, the red-shaded area represents the HV for the bi-objective case, defined by a reference point (i.e., $\boldsymbol{N}$ in Figure 2.2) and the non-dominated solutions (specifically, f^1, f^2, f^3, and f^4). The larger the HV, the better the Pareto frontier $\mathcal{P}$ is, as it is closer to the Utopia point $\boldsymbol{U}$ and covers a larger portion of the objective space.

Uncertainty space. The uncertainty space, as discussed in [86], serves as an indicator of regions with potential non-dominated solutions that merit further exploration. As illustrated in Figure 2.2, the white space representing uncertainty space corresponds to the non-dominated solutions, which can be explored to uncover additional potential Pareto optimal solutions. In Figure 2.2, the green-shaded area, defined by the Utopia point and the Pareto frontier, denotes a region where no non-dominated solutions are currently present. If a non-dominated solution f' were to exist within this green-shaded area, it would dominate all the current Pareto optimal solutions, thereby invalidating their Pareto optimality. Consequently, the uncertainty space is defined by excluding the regions with no non-dominated solutions and subtracting

the HV from the hyper-rectangle formed by the Utopia and reference points. The smaller the uncertainty space, the better the Pareto frontier.

In the experiments discussed in later chapters, we use HV for two key reasons. First, both HV and uncertainty space utilize the dominated space defined by the Pareto frontier and a reference point. Second, since the true Pareto frontier is unknown in our MOO problem and only an approximation of the Pareto frontier is available, the green-shaded area cannot be safely discarded because a true Pareto solution may exist within this region. Consequently, only the dominated space can be discarded with certainty, reducing the uncertainty space to the entire objective space excluding HV. Thus, both methods function similarly in computing HV within the entire objective space.

2.1.4 MOO for SQL Queries

MOO for SQL queries searches for optimal Pareto optimal query plans that achieve optimal trade-off among competing objectives [29, 34, 35, 94, 95, 96, 98, 99, 100].

Given a query modeled as a set of tables to be joined, with the query plan specifying the join order and the operator implementation used for each scan and join, multi-objective query optimization is addressed in [94, 95, 96]. An approximation scheme is proposed to guarantee near-optimal plans by minimizing both execution time and monetary fees [94]. This scheme employs an approximation precision that is iteratively refined until a near-optimal plan is identified. An incremental algorithm is developed to iteratively form the Pareto front through cost-bounded optimization [95]. Plans generated in different iterations are retained and reconsidered for the same query to avoid redundant computation. They further propose a fast algorithm to approximate the Pareto plan set using a multi-objective version of the hill climbing method, which improves upon a random query plan [96]. Similar to [95], a cache of Pareto-optimal plans is maintained to store potentially useful intermediate results discovered across different iterations. A weighted sum model (WSM) is proposed to select the optimal query execution plan by optimizing query execution time, monetary cost, and energy consumption [34]. It introduces a normalized weighted sum algorithm to combine weights for both users and the environment.

However, it only considers the optimal choice from a set of 20 query plans. An extended multi-objective query optimizer is proposed for Spark SQL [29]. The Pareto front is formed by the fastest plans under different fixed computing instances and numbers of Spark executors.

Multi-objective query re-optimization is considered in [98, 99, 100]. Based on the optimal query plan obtained from the WSM [34], ML-based models are applied to determine whether to re-optimize a stage during query execution [98]. These models use the differences in selectivities of a stage before and after execution as inputs. Q-learning is utilized to determine the allocation of machines to each operator by considering execution time and monetary cost in its reward function [99]. The reward is further diminished if the execution time and monetary cost violate the Service Level Agreement (SLA) [100].

MOO for workflow scheduling [35] assigns specific operators to machines (e.g., containers) in the cloud while simultaneously optimizing both time and cost. This 2D optimization problem is addressed by first optimizing a single-objective problem under each container limit (e.g., 20 containers). Pareto solutions are then formed by combining solutions from all container limits, with dominated solutions being filtered out.

These works address a MOO problem distinct from ours, which focuses on tuning parameters of big data systems to optimize multiple performance goals, including latency, monetary cost, and other objectives.

2.2 Parameter tuning of DBMS and big data systems

Parameter tuning is crucial in database management systems (DBMS) and big data systems, as performance metrics such as latency heavily depend on parameter configurations. This area has garnered significant interest and has become an active research field. The following section discusses existing work of parameter tuning in DBMS and big data systems.

Tuning in DBMS. Recent work applies machine learning (ML) techniques to

DBMS tuning, where tuning decisions are made based either on feedback from trial runs or on learning-based models, such as Gaussian Processes (GP) and Reinforcement Learning (RL) models.

Trial runs. UDO [102] aims to tune the parameters of database systems for specific workloads within a given time limit. It categorizes parameters into 'heavy' and 'light' types, where heavy parameters are costly to modify and light parameters are less expensive to adjust. The method proposes using a RL-based approach to first optimize and recommend the heavy parameters in the outer loop, followed by the optimization of light parameters based on the recommendations for heavy parameters. The evaluation is conducted based on feedback from trial runs. However, UDO is not specifically designed to address MOO.

GP-based. ONLINETUNE [114] addresses the online database tuning problem by considering both the dynamic nature of the context and the safety of the recommended configurations. Dynamicity refers to the tuner's ability to adapt to changing contexts, such as varying workloads, while safety of configurations ensures that the recommendations do not degrade the database's performance. To address dynamicity, they propose incorporating the context features along with configurations into the GP model. To address safety of configurations, domain knowledge and the confidence bounds of the contextual GP are utilized to exclude unsafe configurations. OtterTune [97] is designed to optimize DBMS configurations for workloads by utilizing historical data and modeling with GPs to recommend optimal configurations. To address the issue of having a large number of parameters, OtterTune identifies the most important knobs that significantly impact the DBMS's performance and determines their importance order using the Lasso path algorithm [33]. OtterTune also leverages past experience by employing a workload mapping strategy and then uses GPs to recommend configurations for the DBMS. ResTune [113] aims to automatically find the optimal configurations for a database management system by optimizing resource utilization while meeting throughput and latency requirements. The latency and throughput constraints are modeled using GPs, and the BO framework is employed to recommend promising feasible configurations.

RL-based. QTune [53] proposes a query-aware database tuning system that em-

ploys a deep reinforcement learning model for each query. It uses the feature vector of the given query as input to recommend optimal knob values. QTune supports query-level, workload-level, and cluster-level tuning, offering optimization choices for query latency, throughput, or both. CDBTune [110] utilizes deep reinforcement learning to determine the optimal configurations for cloud databases. It relies on a reward function that combines latency and throughput with weighted sums, where the weights are adjusted according to user preferences. The optimal configuration is recommended using the deep deterministic policy gradient method to handle high-dimensional state spaces.

UniTune [112] proposes a unified framework to coordinate existing ML-based tuning agents for DBMS, including both BO-based and RL-based agents. To avoid inaccurate modeling caused by environmental changes after tuning a component, they encapsulate environmental changes as context in the tuning models. For example, they augment the context in the input of the surrogate model used by the BO-based agent. A reinforcement learning algorithm is used to allocate the tuning budget (e.g., time) permitted for tuning a DBMS.

Tuning in big data systems. Techniques used in state-of-the-art works for tuning big data systems primarily fall into two categories: search-based methods and learning-based methods.

Search-based. Bilal et al. [12] introduce a framework for the automatic parameter tuning of stream processing systems, with a focus on Apache Storm. The performance benchmark is a discontinuous objective that includes both latency and throughput. Unlike our approach, which optimizes multiple objectives simultaneously, their method selects candidate configurations based on solutions that achieve at least a minimum level of throughput. A gray-box heuristic approach searches for optimal parameter values by using user-provided hints, which indicate whether changes in parameter values improve latency and throughput. However, such hints are not available in our problem.

BestConfig [119] searches for the optimal system configuration for a given workload, aiming to optimize a scalar performance metric that may include one or multiple performance goals. They combine the divide-and-diverge sampling method

with the recursive bound-and-search algorithm to address the two subproblems of sampling and performance optimization. The sampling method diverges samples by representing each interval of every parameter exactly once within a high-dimensional configuration space, aiming for broad coverage with a limited number of samples. The bound-and-search algorithm then recursively searches for the best configuration within a bounded space defined by these samples.

ML-based. ML-based methods use GP or other learned models to identify more promising configurations.

Tuneful [26] is designed to tune high-dimensional Spark configurations for various data analytical workloads. It aims to achieve results comparable to state-of-the-art tuners while requiring significantly fewer executions. Based on the observation that only a small subset of parameters significantly impacts overall performance, Tuneful focuses on tuning only the important parameters using GP models. Additionally, a similarity analyzer is designed to reuse existing models for new, similar workloads based on their similarities, which greatly reduces tuning costs over time.

LOCAT [108] proposes a BO-based approach to automatically tune Spark SQL applications, with the goal of optimizing execution time. To avoid the high overhead of collecting training samples, they exclude configuration-insensitive queries during the tuning process. Moreover, to reduce optimization time by decreasing the number of tuning parameters, they eliminate unimportant parameters using the Spearman correlation coefficient approach and further identify key parameters through Kernel principal component analysis. Finally, to adapt the tuning process to changes in input data size, they represent each configuration in the surrogate model using both configuration parameters and input data size. They then optimize these configurations within the BO framework.

Recent work [55] proposes an online tuning framework for periodic Spark jobs, supporting multiple tuning goals and constraints. Rather than addressing MOO directly, multiple tuning goals are represented using a single formula with constraints. Preference weights are assigned to runtime and resource functions, and constraints set upper bounds on runtime and resource usage. The optimization is then performed using the BO framework. The GP model serves as the surrogate model

and incorporates data size to support dynamic workloads. Additionally, to enhance accuracy, the tuning history from previous similar tasks is integrated into the surrogate model. To handle the high-dimensional parameter space, a safe configuration acquisition method is designed to explore a subspace of parameters.

RelM [51] addresses memory management tuning for workloads by analyzing and utilizing the interactions between important memory configuration options. The goal is to recommend a memory pool setup that prioritizes safety, high task concurrency, and cache hit ratio, while minimizing garbage collection overheads. This analysis is used to refine the GP surrogate model in BO to identify safe, efficient, and low-overhead configurations.

ClassyTune [118] is designed to automatically tune configurations for cloud systems. Instead of modeling system performance directly, ClassyTune uses a classification model to identify the optimal performance configuration. The classification model predicts whether one configuration outperforms another, generating a list of winning configurations to identify promising subspaces. ClassyTune then samples these promising subspaces and selects the best-performing configuration based on the evaluation of these samples. It is not applicable to our problem, as classification methods cannot be easily extend to MOO settings.

LITE [56] is proposed for auto-tuning Spark configurations. It employs a code learning framework that extracts stage-level code and DAG scheduler information as training features, enabling it to use these code features to learn complex correlations between application performance and knob values. Additionally, the online model identifies knob regions using the mean value of a knob as the center and the standard deviation to determine the span. This approach helps to reduce tuning overhead by randomly sampling a small number of candidates within the promising, smaller search space. However, LITE is not practical for cloud providers because user code is not available due to privacy constraints.

These solutions are not suitable for our MOO problem, as they focus on a scalar performance goal and do not solve the complex control of fine-grained parameters.

2.3 Resource optimization in cloud data analytics

Resource optimization involves determining and optimizing resource configurations for queries, which is crucial in cloud data analytics because the number of resources directly impacts performance (e.g., latency) and monetary cost due to the pay-as-you-go cloud pricing strategy. Based on resource optimization strategies, state-of-the-art approaches can be categorized into three directions: resource saving, optimal resource estimation, and optimal resource configuration tuning.

Resource Saving. The main concept is to prevent resource over-allocation for jobs.

Bag et al. [7] propose a plan-aware resource allocation approach that releases excess resources (i.e., containers) by estimating the peak resource usage for each stage in the remaining job execution. While this method saves resources without impacting job latency by leveraging the complex DAG structures of stages, it does not optimize latency and cost simultaneously.

Observing that default resource allocation is often over-allocated in SCOPE, AutoToken [83] introduces a peak resource allocation strategy to conserve resources for recurring jobs. It develops a model that learns the relationship between required resource usage and job characteristics from past recurring jobs, and then predicts the appropriate resource counts for incoming jobs. However, it only conserves resources without achieving optimal resource allocation for each job.

Optimal Resource Estimation. Recent work proposes modeling the relationship between the number of resources and performance (e.g., job execution time) using historical data, and then estimating the optimal amount of resources on demand.

JUGGLER [1] is designed to cache appropriate datasets (e.g., RDDs in Spark) in memory and recommend an optimal cluster configuration (i.e., number of machines) to optimize execution time and cost (i.e., number of machines $\times$ time) for big data applications. It determines the datasets based on the maximum reduction in computational overhead achieved through caching. Using the size of the cached datasets, JUGGLER predicts the required number of machines based on the memory available for caching per machine. Finally, it selects the most suitable dataset

based on predefined deadlines or cost constraints.

To determine the optimal instance configuration for query processing in the cloud, recent work [52] focuses on optimizing workload cost in dollars under runtime constraints, rather than minimizing running time. They propose an intuitive model with six variables (e.g., CPU hours, scanned data) to represent a query workload. For a given configuration of this model, they estimate workload cost and execution time for different hardware instances and generate a corresponding Pareto frontier, where each Pareto point represents one instance. After estimating all instances, they recommend the instance with the optimal workload cost or execution time as the best hardware configuration. However, this paper focuses solely on recommending the best instance without addressing fine-grained parameter control. Additionally, the proposed model is quite simple, using only six variables, whereas our MOO problem involves tuning thousands of resource parameters.

ReLocag [37] proposes a predictor to allocate a near-optimal number of CPU cores for data-parallel jobs. The inputs to the predictor include the number of CPU cores and contextual information about the jobs, while the output is the predicted job completion time. To train the prediction model effectively, ReLocag introduces a heuristic sampling method that selects training samples in a way that avoids both underestimations and overestimations of the number of CPU cores. Finally, the near-optimal number of CPU cores is determined by selecting the configuration that corresponds to the smallest job completion time, based on inputs of various CPU core counts. However, it addresses a single-objective optimization problem and focuses solely on CPU cores, which is not applicable to our scenario as we need to optimize multiple objectives across a broader range of resource parameters.

WiSeDB [67] proposes generating cost-aware workload schedules using decision-tree models. These schedules determine the number of Virtual Machines (VMs) to provision, their placement for queries, and the order of query execution within the workload. Cost functions are defined by the price of renting VMs, performance goals (e.g., latency of a query template or workload), and penalties incurred for failing to meet these performance goals. A decision tree model is trained using optimal schedules derived from the minimum cost paths in the schedule graph. Given a

workload as input, the decision tree estimates the performance and cost of executing the workload. The model then recommends the schedule that best balances the performance goal and cost. While WiSeDB explores the trade-off between latency and cost for queries in a workload, it does not support finer control of resources within a query.

Morpheus [39] codifies implicit user expectations as Service Level Objectives (SLOs) and enforces these SLOs using scheduling techniques, focusing on job deadlines or completion times. Morpheus aims to economically allocate resources (i.e., containers) for job execution to meet the SLOs. It provides a job model that estimates resource usage at each time step based on historical data. However, its optimization of container allocation uses system utilization as the cost function, which is not suitable for optimizing multiple objectives simultaneously.

Optimal Resource Configuration Tuning. Resource configurations encompass various elements such as resource instances (e.g., VMs, machines, containers), resource managers in database systems, and runtime parameters related to resources (e.g., partition count of a stage). Achieving optimal resource configurations has garnered significant attention.

Accordia [61] is designed for recurring big data analytics jobs and aims to find the optimal cloud configuration for VM instances while minimizing total execution cost and meeting job completion time requirements. It extends the Gaussian-Process (UCB) algorithm for online cloud configuration selection, but it does not support MOO.

CLEO [85] aims to integrate learned latency models of operators within the query optimizer (i.e., a Cascades optimizer) to minimize latency at each stage by tuning the partition count. However, it addresses a single-objective optimization problem rather than MOO. Furthermore, it focuses solely on tuning the partition count, without considering the hundreds of parameters and other resource optimization choices.

Perforator [80] addresses the resource optimization problem by identifying the best hardware resources (e.g., VMs) for a given execution plan. Its goal is to either minimize the execution time of queries or the monetary cost. This single-objective

optimization problem is addressed by exhaustively exploring the search space, which includes cluster capacity, available VM types, and predicted execution time from its model. However, it does not address MOO and this exhaustive search approach is not practically feasible for large parameter spaces.

Tempo [90] tunes configurations of the resource manager in multi-tenant database systems, considering various Service-Level Objectives (SLOs), such as average job response time under a given threshold and job throughput. It addresses a MOO problem for a single SLO across multiple tenants by using a proxy model to transform the constrained MOO into a constrained single-objective optimization problem, with the goal of optimizing the SLO for all tenants. However, this approach does not address complex fine-grained parameter control.

Li et al. [54] addresses resource optimization for parallel database systems through an optimal data partitioning scheme among a set of machines. This partitioning scheme is modeled and optimized as a linear programming problem to minimize workload execution time.

Pimpley et al. [76] [77] propose allocating an optimal number of containers to a query based on runtime as the performance metric at the big data processing platform SCOPE [14]. They train a global model for all incoming jobs to capture the relationship between the number of containers and performance, which is approximated using an exponentially decaying curve. The gradient descent method is then applied to optimize the resource allocation. AutoExecutor [84] employs a similar approach to model the resource-performance relationship for Spark. It extends Spark to allocate resources at the beginning of a query execution, making the resource allocation transparent to users. However, their approach focuses solely on single-objective optimization.

Overall, this line of work focuses on a small set of resource-related parameters, whereas our cloud optimizer deals with a large number of parameters.

2.4 Summary

This chapter surveys the state-of-the-art work on multi-objective optimization (MOO), parameter tuning, and resource optimization in big data systems. However, none of

these approaches address the specific challenges of our problem, which include multiple competing performance goals and budgetary constraints, a high-dimensional parameter space with complex control, and the need for high efficiency in cloud environments (e.g., 1-2 seconds). This thesis will propose novel MOO approaches for Alibaba Cloud and Spark, addressing the technical challenges outlined above.

CHAPTER 3

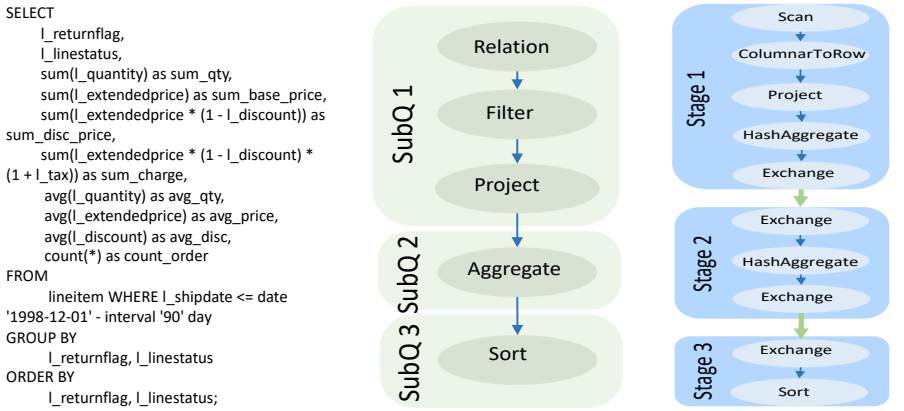
System Overview and Problem Statement

This chapter presents the system overview of the *Cloud Optimizer*, detailing the job description of an incoming analytical task and the system design based on Multi-Objective Optimization (MOO). It also introduces a general problem statement of MOO within the *Cloud Optimizer* framework and provides a formal representation.

3.1 System Overview

3.1.1 Job Description

A cloud analytical task can be modeled as a dataflow program, essentially a Directed Acyclic Graph (DAG) of data collections flowing between operations. Figure 3.1 illustrates an example of an input request for an analytical task, using SQL query Q1 from the TPCH benchmark [92]. Figure 3.1(a) presents the SQL script, while Figures 3.1(b) and 3.1(c) depict the corresponding *Logical Query Plan* (LQP) and *Physical Query Plan* (PQP) after parsing, analysis, and optimization of the submitted SQL query. The LQP represents a DAG of logical operators, while the PQP represents a DAG of physical operators after applying join algorithms in a LQP. The PQP is further divided into a DAG of *stages*, where each *stage* is a DAG of operators with operations such as shuffling or broadcasting serving as boundaries. Both the LQP and PQP can be considered as the dataflow program denoted as a *job*. Similar to the *stage* in PQP, we introduce a new notation, *subQ*, which denotes a group of



(a) SQL Script (b) Logical query plan (LQP) (c) Physical query plan (PQP)

Figure 3.1: An example of an analytical task of TPCH-Q1

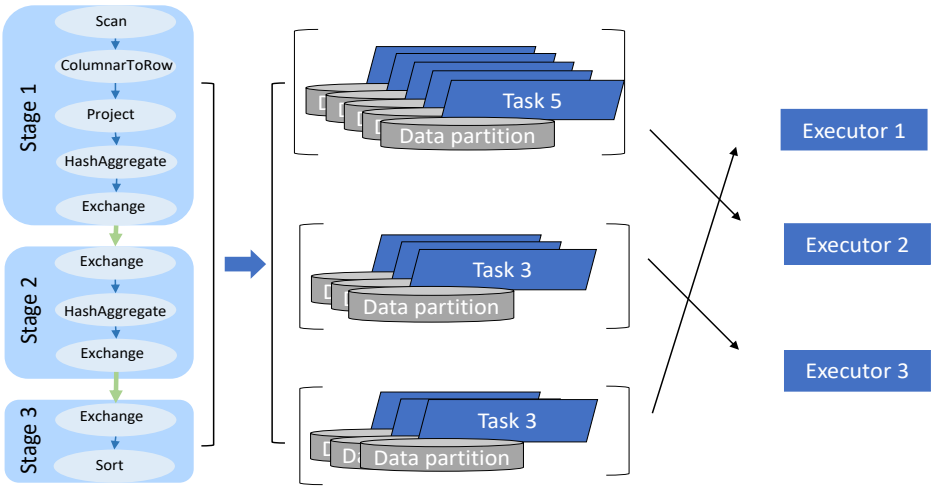


Figure 3.2: Job description

logical operators that correspond to a *stage* when the logical plan is translated to a physical plan.

Figure 3.2 illustrates an example of a *job* with a PQP, where each *stage* itself is composed of a DAG of *operators*. Dependencies exist between stages, indicating that a child stage cannot start running until all of its parent stages finish. Within each stage, computations are further broken down into *tasks*, which operate on different data partitions. The number of partitions and their sizes are determined by corresponding *parameters* of the big data systems. These *tasks* are executed on *executors*, with the number of *executors* and the allocation of CPU cores and memory size also being determined by *parameters*.

The control of parameters in big data systems is intricate, with varying behaviors

observed even within stages and across different systems. For example, in Spark [109], resource settings must be uniform across all stages, while factors such as partition size and the number of partitions may differ between stages. Conversely, in MaxCompute [71] within Alibaba Cloud, there is support for setting the number of CPU cores and memory size for each data partition independently. In response to these nuanced control behaviors, this thesis proposes optimization solutions tailored to Spark and MaxCompute, as detailed in Chapters 5 and 6 respectively.

3.1.2 System Design

We assume that each analytical task comes with user-specified objectives that need to be optimized during execution. In order to execute these analytical tasks effectively, it is essential to instantiate parameters that govern system behaviors. These parameters may include settings related to resource allocation (such as the number of CPU cores and memory size), degree of parallelism, join strategies, compression options, shuffling strategies, and more.

The configurations of cloud analytical tasks need to be quickly adapted when objectives or loads change, imposing requirements for high efficiency. Building upon our prior work [86], our system is designed with two asynchronous modules: a model server providing objective predictions and a MOO procedure computing and recommending optimal configurations within a few seconds. To avoid the time-consuming modeling process of learned models, the model server is trained offline whenever new training data becomes available. MOO is performed online, leveraging the latest trained model to return objective predictions by taking configurations as input.

Our system design offers two significant benefits. Firstly, our MOO module is designed to work with multiple model types, including complex ones such as Deep Neural Networks (DNNs), Gaussian Processes Regression (GPR), Multi-layer Perceptron (MLP), as well as simple closed-form regression functions. This stands in contrast to existing works like [97, 110], which are limited to specific model choices (GPR or DNN). Our approach provides users with a more flexible way to implement different models. Secondly, our system supports any MOO algorithm for analytical

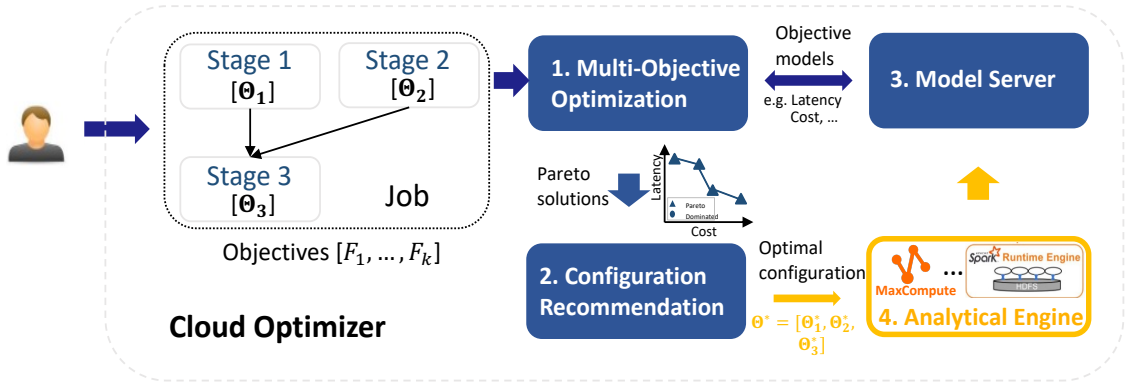


Figure 3.3: System design

tasks, catering to the diverse requirements of users. As a result, our system design offers flexible choices for both model servers and MOO approaches, meeting the varying demands of users.

Figure 3.3 illustrates the architecture of our *Cloud Optimizer*, which interfaces with existing big data systems (like Spark [109] and MaxCompute [71]) as analytical engines. When a user submits a *job*, which is essentially a dataflow program structured as a DAG, along with a set of objectives $(F_1, \ldots, F_k)$, each *stage* of the job can configure its parameter values (e.g., $\boldsymbol{\Theta_1}$, $\boldsymbol{\Theta_2}$, $\boldsymbol{\Theta_3}$). Our *Cloud Optimizer* is versatile and accommodates various objectives, including but not limited to latency, monetary cost, and IO cost. Additionally, it offers flexibility for incorporating additional objectives based on users' specific requirements.

Our *Cloud Optimizer* operates on each job in the following manner: Upon receiving a submitted *job* along with multiple user-specified objectives, the MOO module takes them as input. It employs MOO algorithms and solvers to produce a set of Pareto solutions. These Pareto solutions are then processed by the recommendation module, which selects the optimal configuration based on user preferences. This configuration, designed to best explore all specified objectives, is subsequently executed on analytical engines such as MaxCompute and Spark. The model server plays a crucial role in providing objective predictions during the optimization process. These predictions are utilized by the MOO module to explore the configuration space and compute Pareto optimal solutions. The model server is trained offline and updated whenever new training data becomes available. It collects runtime data during job execution, including query traces with query plans and operators, as well as

machine-level traces such as machine system state. Separate predictive models are employed for each objective.

Modeling choices. Although this thesis primarily focuses on MOO, it is important to briefly discuss relevant modeling techniques, because they provide objective predictions during the optimization process. Our optimizer provides support for various modeling options: *Simple models* where users can define customized models based on domain knowledge, employing simple function shapes. For instance, cost can be represented as resource usage (e.g., CPU cores), the monetary cost of resource utilization (e.g., CPU cores multiplied by latency), or a weighted combination of CPU-hour and IO cost [5]. *Learned models*, where existing research has applied Machine Learning (ML) methods [68, 112], Reinforcement Learning (RL) methods [112], or Gaussian Processes Regression (GPR) [114] to learn predictive models. Chapter 5 and Chapter 6 will provide an overview of the model construction based on a large number of parameters tailored to two specific big data systems.

3.2 Problem Statement of MOO

The MOO module within our *Cloud Optimizer* is responsible for handling the optimization problem and implementing various MOO approaches. We define the problem statement for the general MOO problem within this module as follows:

Definition 3.2.1. General MOO problem in Big Data Systems

$$\begin{aligned} \arg\min_{\boldsymbol{\Theta}} \quad & \boldsymbol{F}(\boldsymbol{\Theta}) = \begin{bmatrix} F_1(\boldsymbol{\Theta}) = f_1(\boldsymbol{\Theta}, \alpha) \\ \vdots \\ F_k(\boldsymbol{\Theta}) = f_k(\boldsymbol{\Theta}, \alpha) \end{bmatrix} \\ s.t. \quad & \boldsymbol{\Theta} \in \Sigma \subseteq \mathbb{R}^d \end{aligned} \tag{3.1}$$

where $F_1, \ldots F_k$ denote k objectives to be optimized. $f_1, \ldots f_k$ represent the evaluation functions used to obtain objective values. $\boldsymbol{\Theta}$ denotes the configuration to be optimized and α denotes the non-decision variables.

In our optimization problem, $f_1, \ldots, f_k$ functions could involve learned models

or aggregations based on such models, with varying granularities of learned models. For instance, consider the optimization problem of a query (i.e., a job with a DAG of *stages*) with objectives like query-level latency and query-level cost. When using a coarse-grained learned model to predict the objective value of an analytical task, where $f_1, \ldots, f_k$ represent objectives of a query, these predictions can be computed directly with configurations during optimization. However, when employing finer-grained models for better control over each *stage*, with each learned model predicting objective values of a *stage*, the overall query-level latency and cost must be aggregated (e.g., using `sum`) from multiple stage-level latency and cost predictions.

Chapter 5 will expand the configurations and learned models for resource optimization of practical workloads in Alibaba Cloud, aiming to achieve Pareto optimal solutions for each *stage*. The *configurations* of each instance in a *stage* can vary, including 1) CPU cores and 2) memory size of each instance. Thus, given m instances within a *stage*, the total number of configurations (i.e., $\boldsymbol{\Theta}$) is $2 \times m$, introducing a high-dimensional parameter space since m could be in the thousands. Fine-grained models are applied to predict instance-level latency and cost. The values of stage-level objectives (i.e., $F_1, \ldots, F_k$) are aggregated from the instance-level values, where the stage-level latency is the maximum latency among all instances, and the stage-level cost is the sum of all instance-level costs within a stage, defining $f_1, \ldots, f_k$. The goal is to minimize the stage-level latency and cost.

Chapter 6 will address the MOO problem defined in Equation (3.1) with finer-granularity control of Spark parameters, aiming to minimize the query-level latency and cost at compile-time (i.e. with a DAG of *subQs*). Spark supports Adaptive Query Execution (AQE) to enable re-optimization of a logical plan of a *stage* at runtime with actual statistics. Some parameters can be tuned independently for each query stage (i.e. $\boldsymbol{\theta_p}$), while others are shared across all stages (i.e. $\boldsymbol{\theta_c}$), introducing a high-dimensional parameter space and complex constraints. Thus, the configurations of different *subQs* could vary in $\boldsymbol{\theta_p}$ and are subject to the constraint of identical $\boldsymbol{\theta_c}$. Fine-grained models of each *subQ* are built to obtain the subQ-level objective values, i.e., analytical latency and cost. The query-level objective values (i.e., $F_1, \ldots, F_k$) are then aggregated from the subQ-level objective values with the

constraint of identical $\boldsymbol{\theta_c}$, which defines $f_1, \ldots, f_k$. The configurations of a query (i.e., $\boldsymbol{\Theta}$) include the configurations of all *subQs*.

3.3 User Experience

During big data processing, users can specify performance goals, hard constraints and optionally preferences to enhance system efficiency. Users input dataflows into the *Cloud Optimizer* on cloud platforms, where they may set multiple, possibly competing, performance objectives, such as latency and cost. Additionally, users define hard constraints; for example, a job must complete within one day. Preferences are specified optionally. For instance, while users generally seek a balance between latency and cost, assigning equal weights [0.5,0.5] preferences might shift based on the time of day. During peak hours, greater emphasis may be placed on reducing latency, with weights adjusted to [0.7,0.3]. Conversely, during off-peak hours, cost efficiency becomes the priority, with a weight distribution of [0.3,0.7].

Addressing MOO challenges within the *Cloud Optimizer* diverges significantly from traditional approaches that employ fixed weights to transform these into a single-objective optimization problem. Prior work [53, 110, 119] used fixed weights to combine multiple objectives into a **single objective** (SO) and solved it to return one solution, denoted as the `SO-FW` method. It is a special case of a classical MOO algorithm, *weighted sum* (WS) [69], that repeatedly applies n weight vectors to create a set of SO problems and returns a solution for each, denoted as the `MO-WS` method. It is known from the theory of WS that trying different weights to create SO problems is unlikely to return points that evenly cover the Pareto front unless the objective functions have a very peculiar shape [69]. Empirically, `MO-WS` has been reported with sparse coverage of the Pareto front in prior work [86]. Such sparse coverage of the Pareto front leads to poor adaptability when the user shifts preference (e.g., from favoring latency to favoring cost) because there are not enough points on the Pareto front to capture the tradeoffs between the objectives.

For this reason, this thesis casts the optimization problem in *multi-objective optimization* (MOO) framework [23, 69, 73, 74], which computes the Pareto front properly to capture the tradeoffs and later allows us to recommend one that best

matches the user preference.

3.4 Summary

Our *Cloud Optimizer* seamlessly integrates MOO with a model learning module to dynamically recommend optimal configurations for analytical tasks within seconds. Given that learned models are trained offline and the model server can continually update models with new training data in the background, the efficiency of generating Pareto solutions from MOO becomes a critical performance metric. Chapter 5 and Chapter 6 will introduce novel MOO approaches tailored to their respective big data systems.

CHAPTER 4

Benchmark Study of Multi-Objective Optimization

This chapter introduces a benchmarking analysis of existing MOO methods and solvers, identifying their limitations, particularly in terms of efficiency and the quality of Pareto solutions, when applied to cloud optimization.

The optimization problem defined in Equation (3.1) is a Mixed Integer Non-Linear Programming (MINLP) problem due to its non-linear objective functions and mixed parameter types, including both continuous and integer variables. Especially, the objective functions in our optimization problem are learned models, such as Deep Neural Networks (DNN), Gaussian Process Regression (GPR), and Multi-layer Perceptrons (MLP), which utilize non-linear functions. For example, the Rectified Linear Unit (ReLU) is a non-linear activation function used in DNNs.

Solvers and Multi-Objective Optimization (MOO) algorithms are essential implementation and theoretical tools for solving MINLP problems with multiple objectives. Theoretically, an MOO problem can be approached in two ways: transforming it into a single-objective optimization problem, or applying MOO algorithms directly. The former approach can be optimized using various solvers. Examples of using such transformation techniques include Weighted Sum (WS) [69], Normalized Normal Constraint (NNC) [74] and Progressive Frontier (PF) [86] algorithms. The other direction solves MOO directly using MOO algorithms, such as NSGA-II [22].

To better understand how existing solvers and MOO algorithms perform for our

optimization problem, the following sections will provide a study on existing solvers and a benchmarking analysis of MOO algorithms.

The objective function in our MINLP problem utilizes a DNN with ReLU as the activation function, serving as an example of learned models.

Example learned model. In this context, a DNN with ReLU activation functions can be represented as a function f mapping a mixed-integer variable vector X to an objective y. Formally, this can be described as: $y = f(X)$, where $X = [x_1, x_2, ..., x_n]$ is a vector of mixed variables, and each x_i can be either a continuous or an integer variable.

A simple DNN with one layer and two input variables is defined as follows:

$$y = ReLU(XW^T) \tag{4.1}$$

where $X = [x_1, x_2]$ is the variable vector, $W = [w_1, w_2]$ are the weight vector.

Then, a two-layer mapping with two input variables is represented as follows:

$$y = ReLU(Z(W^2)^T), \quad \text{where} \quad Z = ReLU(X(W^1)^T) \tag{4.2}$$

where Z is the intermediate results of the first layer, X is the variable vector, W^1 and W^2 are the weight vectors of the first and second layers respectively, defined in Equation (4.3).

$$W^1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \end{bmatrix} \quad W^2 = [w_1^2, w_2^2] \tag{4.3}$$

More complex DNN can be built in the same manner.

4.1 Study on Solvers

This section reviews existing solvers for single-objective optimization. Theoretically, this MINLP problem can be tackled directly using a Non-Linear Programming (NLP) solver or, if the MINLP problem can be converted into a Linear Programming (LP) problem, an LP solver. This section also analyzes how the commercial solver

Knitro [42] addresses the MINLP problem, specifically focusing on the cases where the objective function is represented by a DNN with ReLU activation functions.

4.1.1 Existing solvers

This thesis examines two basic solvers, i.e. random sampling and grid search, as well as existing NLP solvers capable of addressing our MINLP problem.

Basic solvers. Random sampling and grid search are two fundamental solvers feasible for addressing our MINLP problem.

Random sampling generates independent samples randomly from a uniform density over the same parameter space as the initial set of trials, with the best sample selected as the solver's solution. While this method can be efficient, it only produces a limited number of random samples from the parameter space.

Grid search requires a set of grid values for each variable, denoted as $N^{(1)}, \ldots, N^{(n)}$, and constructs the initial set of trials through the *Cartesian Product* of these n sets of grid values [10], i.e., $S = \prod_{i=1}^{n} N^{(i)}$. The best solution is subsequently selected from this initial set of evaluations. Grid search ensures even coverage of the parameter space by establishing uniform grids for each variable. However, this *Cartesian Product* approach is subject to the curse of dimensionality, as the number of trials grows exponentially with an increase in the number of variables.

Both solvers are basic and straightforward to implement and parallelize. However, these methods do not guarantee optimality of the solutions obtained.

NLP solvers. Methods for addressing MINLP typically involve the decomposition or relaxation of the problem into simpler subproblems, usually consisting of NLP and MILP components. Consequently, reliable and efficient NLP and MILP solvers are often essential components of effective MINLP solutions [18]. In examining a general MINLP problem, this thesis reviews existing NLP solvers, assessing their capabilities based on information from user manuals and webpages.

SCIP [11, 82] is a non-commercial solver that tackles MINLP problems using a spatial branch-and-bound algorithm. However, SCIP cannot solve our specific MINLP problem, as it lacks support for non-linear function representations incorporating the `max` function, which is utilized in the DNN model.

Commercial solvers like BARON [8], Artelys Knitro [42], and LindoGlobal [58] are designed for achieving global optimal solutions of MINLP problems, employing various branch-and-bound type algorithms. Among these, BARON, unlike Knitro and LindoGlobal, does not support certain functions, including `max`, making it unsuitable for our MINLP problem.

The subsequent section will use Artelys Knitro [42] as a representative example of NLP solvers to illustrate how our MINLP problem is addressed.

4.1.2 Knitro

Artelys Knitro [42] is a commercial optimization software library designed primarily for solving nonlinear optimization problems. The library offers four algorithms for NLP and two algorithms specifically for MINLP [45, 46].

The optimization problem addressed by Knitro is formulated as follows:

$$\min f(x) \quad \text{subject to} \quad c^L \leq c(x) \leq c^U \quad b^L \leq x \leq b^U \tag{4.4}$$

Where $f(x)$ represents the objective function, c^L and c^U denote the lower and upper bounds (which may be infinite) on the general constraints, and b^L and b^U represent the lower and upper bounds (which may also be infinite) on the variables. The variables $x \in \mathbb{R}^n$ can be continuous, binary, or integer. The objective function $f(x)$ supports various mathematical functions, such as exponential functions and maximization.

The general implementation of Knitro involves configuring properties for the objective functions, variables, and constraints. These properties include representation functions for objectives, variables, and constraints; the number of objectives, variables, and constraints; the types of variables (e.g., binary, continuous, integer); and the lower and upper bounds for both variables and constraints, among other factors.

To use Knitro for solving our MINLP problem, the objective function must be represented using native mathematical functions. In our example, the learned models use a maximization function within ReLU to represent the objective.

DNN representation using mathematical functions. Given the simple DNN

defined by Equation (4.1) as the objective function, the MINLP can be represented as follows:

$$\min \quad y = ReLU(XW^T) = \max\left(x_1 w_1 + x_2 w_2, 0\right) \tag{4.5}$$

Similarly, for a more complex DNN defined by Equation (4.2) as the objective function, the MINLP is formulated as follows:

$$\min \quad y = ReLU(Z(W^2)^T) = \max(z_1 w_1^2 + z_2 w_2^2, 0) \tag{4.6}$$

where $Z = [z_1, z_2]$ is flattened as:

$$\begin{aligned} z_1 &= \max(x_1 w_{11}^1 + x_2 w_{12}^1, 0) \\ z_2 &= \max(x_1 w_{21}^1 + x_2 w_{22}^1, 0) \end{aligned} \tag{4.7}$$

Knitro recommends setting exact derivatives whenever possible to achieve faster solutions and enhanced robustness [43, 44]. If it is not possible, it offers two types of derivative approximation methods for computing first derivatives: forward finite differences and centered finite differences. These two options assist in optimizing the objective with a DNN when the first derivatives do not exist at points where ReLU is zero.

To enable the solution of our MINLP problem using not only NLP solvers but also Linear Programming (LP) solvers, we provide a theoretical analysis demonstrating the compatibility of our MINLP formulation with an LP solver, such as CPLEX [16]. Further details are provided in Appendix A.1.

4.2 Experimental Evaluation of Solvers and MOO Algorithms

This section presents the experimental results of our optimization problem, evaluating various solvers and MOO algorithms.

Table 4.1: Selected Spark parameters

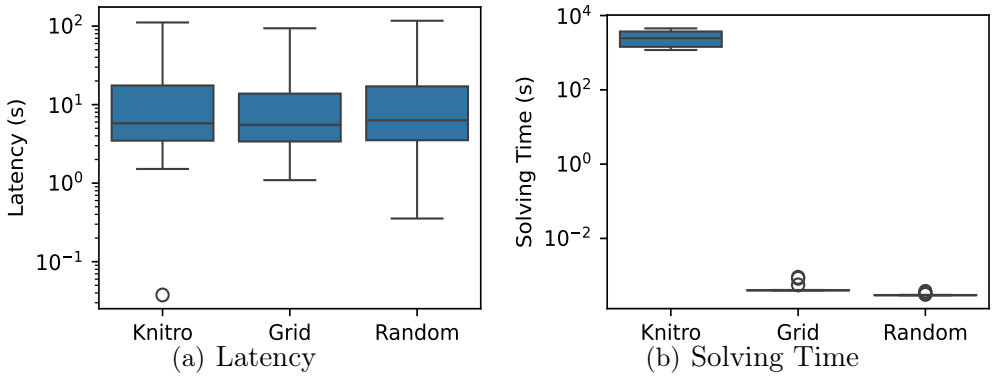
Index	Description
1	spark.default.parallelism
2	spark.executor.instances
3	spark.executor.cores
4	spark.executor.memory
5	spark.reducer.maxSizeInFlight
6	spark.shuffle.sort.bypassMergeThreshold
7	spark.shuffle.compress
8	spark.memory.fraction
9	spark.sql.inMemoryColumnarStorage.batchSize
10	spark.sql.files.maxPartitionBytes
11	spark.sql.autoBroadcastJoinThreshold
12	spark.sql.shuffle.partitions

4.2.1 Setup

Workloads. We apply the batch workloads from our previous work [86]. Our batch workloads use the TPCx-BB benchmark [93] with a scale factor of 100GB. TPCx-BB consists of 30 templates, including 14 SQL queries, 11 SQL queries with UDFs, and 5 ML tasks, which we modified to run on Spark. We parameterized these 30 templates to create 258 workloads. For performance testing, we use 30 workloads to evaluate solvers and 258 workloads to assess the performance of MOO algorithms. We executed these under various configurations, generating a total of 24560 traces, each with 360 runtime metrics. These traces were utilized to train workload-specific models for latency, cost, and other factors. Feature selection identified 12 key Spark parameters, including the number of executors, number of cores per executor, memory per executor, shuffle compression, and parallelism, among others. The complete list is provided in Table 4.1.

Hardware. Our system was deployed on a cluster with 20 compute nodes. The compute nodes are CentOS based with 2xIntel Xeon Gold 6130 processors and 16 cores each, 768GB of memory, and RAID disks.

Solvers and MOO algorithms. To evaluate the performance of the solvers, we compare Knitro with random sampling and grid search, as Knitro solves MINLP



(a) Latency (b) Solving Time

Figure 4.1: Comparison for performance of solvers among 30 workloads

problems directly.

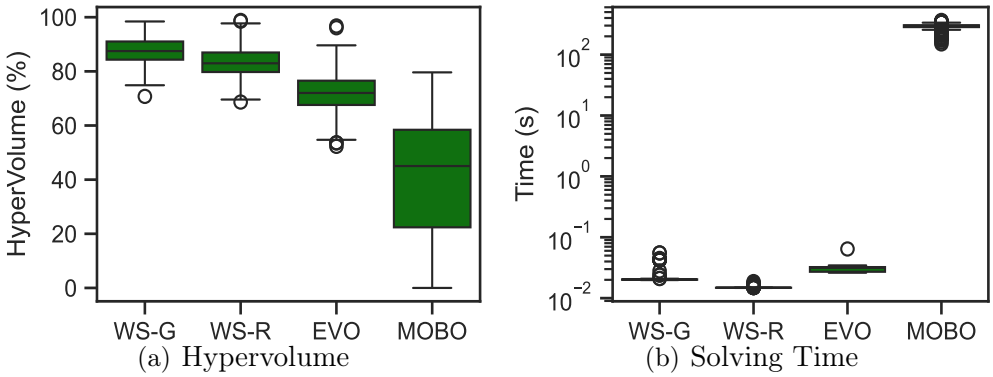
For MOO algorithms, we compare various methods from different categories discussed in Section 2.1.2. This includes the Weighted Sum (WS) [69], NSGA-II [22] as a prominent Evolutionary (EVO) method [23] and qEHVI from BoTorch [20] as a recent Multi-objective Bayesian Optimization (MOBO) method.

Model. We use the DNN model as the latency objective function to evaluate the solvers. For the MOO benchmark evaluation, latency and cost are considered as the two objectives. Cost is defined as the total number of cores, calculated by multiplying the number of executors (index 2 in Table 4.1) by the number of cores per executor (index 3 in Table 4.1).

4.2.2 Evalution Results

Expt 1: *Comparison with existing solvers.* Considering latency as the objective function to be minimized, Figures 4.1 compare the performance of three solvers: Knitro, grid search, and random sampling. Both grid search and random sampling use the same number of samples, totaling 6,720.

Figures 4.1 illustrates that while the average latency across 30 workloads achieved by the random-sampling and grid-search solvers is slightly better than that of Knitro (15.6 seconds and 13.5 seconds for random sampling and grid search, respectively, compared to 15.8 seconds for Knitro), there is significant variation in solving time. Specifically, random sampling achieves lower latency than Knitro for 16 out of 30 workloads, yet its solving time is approximately 1 million times faster than that of



(a) Hypervolume (b) Solving Time

Figure 4.2: Comparison for performance of existing MOO methods among 258 workloads

Knitro.

Expt 2: *Comparison with existing MOO methods.* Figures 4.2 compare the performance of existing MOO methods in terms of hypervolume and solving time. In these comparisons, `WS-R` and `WS-G` refer to the Weighted Sum (WS) method using random sampling and grid search as solvers, respectively.

Compared to `MOBO`, both `WS` and `EVO` achieve higher hypervolume scores (greater than 72.2% vs. 41.8% on average) and significantly lower average solving times (less than 0.03 seconds vs. 290 seconds). Among these, `EVO` performs slightly worse than `WS` in both hypervolume and solving time.

4.2.3 Discussion on solvers

In the previous evaluation, Knitro exhibited lower solution quality compared to basic solvers such as random sampling and grid search. The possible reasons for this discrepancy are outlined below.

Knitro offers multiple tunable options that can influence solver performance, including algorithm settings and warm-start capabilities. Theoretically, solving an MINLP problem with Knitro involves addressing relaxed, continuous nonlinear optimization subproblems, a process governed by the MINLP and NLP algorithms available within the solver. Specifically, Knitro provides two algorithms for MINLP and four for NLP, automatically selecting one method for each by default. Furthermore, Knitro's warm-start strategy allows users to provide an initial point close to the likely solution to facilitate quicker convergence. However, as the tuning of

Knitro is beyond the scope of this thesis, the solver was primarily utilized with its default settings, which may impact performance.

The complexity of the DNN model significantly impacts solver performance. As noted in Section 4.1.2, Knitro advises users to provide exact derivatives to enhance efficiency and robustness. Relying on first derivative approximations can diminish both the performance of the solver and the likelihood of converging to a solution [43, 44]. In our MINLP problem, where the objective function is a complex DNN model, computing both first and second derivatives is challenging due to the application of the chain rule. Additionally, the non-existence of first derivatives at points where the ReLU activation function is zero necessitates the use of first derivative approximations in Knitro, potentially degrading its performance.

The accuracy of predictive models significantly impacts the evaluation results of solvers. In our optimization problem, the objectives are modeled by DNNs. For example, the predictive model tends to underestimate objective values when it receives inputs at the boundary values of variables (i.e., at their lower or upper limits). Given that the objective function is non-convex, Knitro might become trapped in a local optimum. In contrast, basic solvers like random sampling and grid search are more likely to sample boundary values and evaluate them using the predictive model, which could yield better solutions than those obtained with Knitro. Consequently, in our solver evaluation, these basic solvers outperform Knitro.

4.3 Summary

This chapter presents a benchmarking analysis of MOO, including existing solvers and MOO algorithms. Given the high quality and efficiency requirements (i.e., 1-2 seconds) for the *Cloud Optimizer*, Knitro and MOBO are not suitable for cloud use. Consequently, we selected random sampling and grid search as solvers, and WS and EVO methods as MOO baselines for the experiments in Chapter 5 and Chapter 6.

CHAPTER 5

Stage-level Optimizer

This chapter presents MOO algorithms designed to compute Pareto optimal solutions for query stages, which are units defined by shuffle boundaries. In production-scale big data processing, each stage operates with a high-dimensional parameter space, with thousands of parallel instances. Each instance requires resource parameters determined upon assignment to one of thousands of machines, as exemplified by systems like MaxCompute. To achieve Pareto optimality for each query stage, we propose a novel hierarchical MOO approach. This method decomposes the stage-level MOO problem into multiple parallel instance-level MOO problems and efficiently derives stage-level MOO solutions from instance-level MOO solutions. Evaluation results using production workloads demonstrate that our hierarchical MOO approach outperforms existing MOO methods by 4% to 77% in terms of latency performance and up to 48% in cost reduction while operating within 0.02 to 0.24 seconds compared to current optimizers and schedulers.

5.1 Problem Statement and Overview

In this section, we provide background on MaxCompute [71], Alibaba Cloud's big data query processing system, and introduce our proposed extended architecture for resource optimization. We then formally define the Multi-Objective Resource Optimization (MORO) problem and present an overview of our two-step optimization approach.

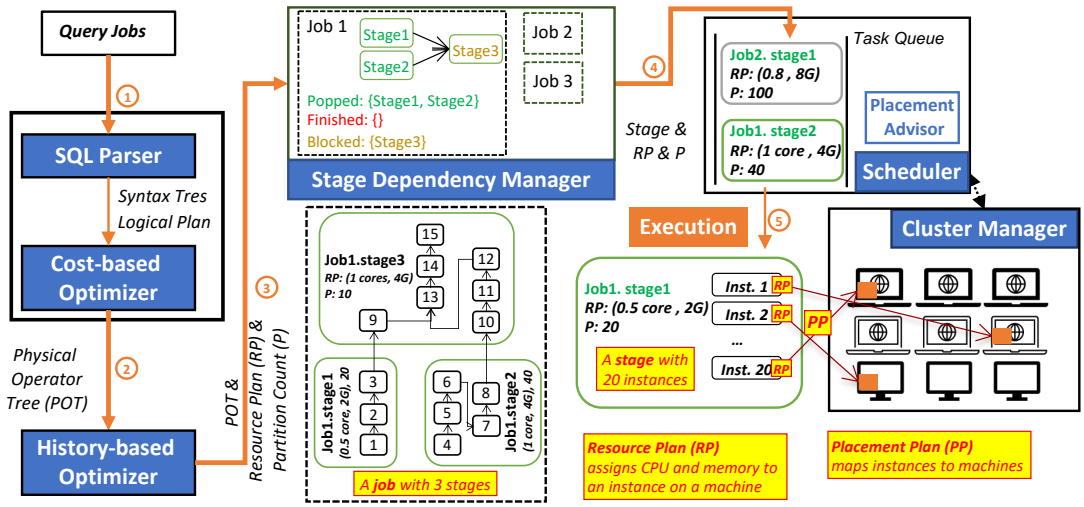


Figure 5.1: The lifecycle of a query job in MaxCompute

5.1.1 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 Figure 5.1, 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.

Figure 5.1 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 [30]. 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 [115]. 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 Figure 5.3.

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 [87], containers with the same resource plan could perform differently on machines with different hardware or system states, making the running environment more complex.

5.1.2 System Design for Resource Optimization

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

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

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

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

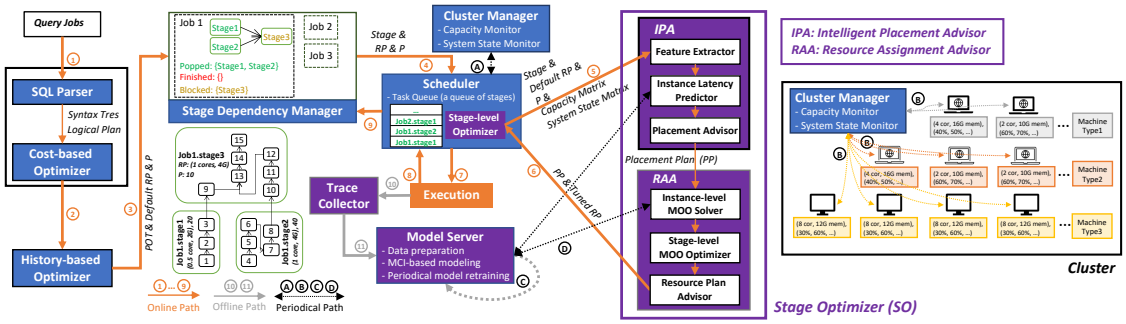


Figure 5.2: Extended system architecture for resource optimization

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 leverage 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.

Figure 5.2 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

Fuxi => Stage latency = 24s
1) Key resource type: CPU
2) Pick machines: m_1, m_2
3) Assign instances: $i_1 \rightarrow m_1, i_2 \rightarrow m_2$

IPA => Stage latency = 16s
1) Predict the L matrix
2) Compute the BPL list: $i_1 = 8s, i_2 = 16s$
3) Assign the instance with the largest BPL: $i_2 \rightarrow m_1$
4) Update the BPL list: $i_1 = 10s$
5) continue 3 until BPL list is empty: $i_1 \rightarrow m_3$

L	m_1 CPU: 40% IO: busy	m_2 CPU: 60% IO: busy	m_3 CPU: 80% IO: idle
i_1 (100 rows)	8s (i_1)	12s	10s (i_1)
i_2 (200 rows)	16s (i_2)	24s (i_2)	20s

Figure 5.3: Example of IPA

latency and then the RP by minimizing both stage latency and cost.

5.1.3 Our Resource Optimization Approach

We next introduce our approach for the *Stage-level Optimizer* 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 [115] decides a PP for m instances as follows: (1) Identify the key resource (bottleneck) in the current cluster, e.g., CPU or IO. (2) Pick m machines with top-m lowest resource watermarks. (3) Assign instances, in order of their instance id, to the m machines, and use the same resource plan for each instance as suggested by HBO. However, its negligence of latency variance among instances leads to suboptimal decisions for PP and RP:

Example 1. Figure 5.3 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) [64]. 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. This assignment is based on the *best possible latency* (BPL) of instances, where BPL is defined as the minimum latency that an instance can achieve among all available machines. 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.

MOO Problems. 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.

Given the technical context provided by the work from our colleague, the learned model f utilizes five channels to characterize various factors as input. Specifically, **Channel 1** introduces stage-oriented features shared among instances within a stage. **Channels 2-5** characterize individual instances: **Channel 2** includes instance metadata such as input row number and size, **Channel 3** details the resource plan for each instance, specifying CPU cores and memory size, **Channel 4** records machine system state including CPU utilization, memory utilization, and IO activities, and **Channel 5** identifies hardware types using machine models to distinguish between different sets of hardware types.

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 Channel 1 and Channel 2 in its multi-channel representation, and $\tilde{y}_j$ to denote the features of Channel 4 and Channel 5 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 $\forall i \in [1, \ldots, m], \Theta_i \in \Sigma_i^* \subseteq \mathbb{R}^d$, 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 5.1.1. Stage-Level MOO Problem.

$$\arg\min_{B,\Theta} \quad \begin{bmatrix} L(B,\Theta) = \max_{i,j} B_{i,j} f(\tilde{x}_i, \Theta_i, \tilde{y}_j) \\ C(B,\Theta) = \sum_{i,j} B_{i,j} f(\tilde{x}_i, \Theta_i, \tilde{y}_j)(\mathbf{w} \cdot \Theta_i^T) \\ \ldots \end{bmatrix} \tag{5.1}$$

$$\begin{aligned} & B_{i,j} \in \{0,1\}, \ \ \forall i = 1...m, \forall j = 1...n \\ s.t. \quad & \textstyle\sum_j B_{i,j} = 1, \ \ \forall i = 1...m \\ & \textstyle\sum_i B_{i,j}\Theta_i^1 \le U_j^1, \ \ldots, \ \sum_i B_{i,j}\Theta_i^d \le U_j^d \ , \forall j = 1...n \end{aligned}$$

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 [23, 69, 74, 86] 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 Definition 5.1.1. 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 Definition 5.1.1 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 Definition 5.1.1 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.

5.2 Formulations of Resource Optimization

Based on the two-step method discussed in the previous section, the Multi-Objective Resource Optimization (MORO) problem can be solved in two scenarios:

1. **Plan A: Optimization over B and Θ.** This scenario considers the entire parameter space with B and Θ as variables, where B determines the placement plan of instances on machines, and Θ determines the resource plans of all instances. **Plan A** solves the MORO by jointly configuring the placement plan and resource plans for all instances.

2. **Plan B: Optimization over Θ.** Based on the optimal placement plan B^*, **Plan B** solves the MORO by configuring the resource plans for all instances in Step 2 (RAA), with Θ as the variable.

5.2.1 Plan A: Optimization over B and Θ

Plan A can be solved by strictly applying the MOO definition in Equation (5.1), which optimizes both B and Θ simultaneously. However, it faces challenges due to the large number of variables and constraints, resulting in $m * (n + d)$ variables and $m + (d + 1) * n$ constraints, where m represents the number of instances, n represents the number of machines, and d represents the dimensions of the resource plans. Given that both m and n can be several thousand, the problem may involve millions of parameters and thousands of constraints.

$$
\begin{aligned}
\arg\min_{B',\Theta'} & \begin{bmatrix} L'(B',\Theta') = \max_{i,j} \mathcal{I}(B'_{i,j}) f(\tilde{x'}_i, \Theta'_{i,j}, \tilde{y'}_j) \\ C'(B',\Theta') = \sum_{i,j} B'_{i,j} f(\tilde{x'}_i, \Theta'_{i,j}, \tilde{y'}_j)(\mathbf{w} \cdot \Theta'^{T}_{i,j}) \\ \dots \end{bmatrix} \\
s.t. & \sum_j B'_{i,j} = |X'_i|, \ \ \forall i = 1...m' \\
& \sum_i B'_{i,j} {\Theta'_{i,j}}^1 \leq \sum_r U_r^1 \ \ \forall j = 1...n', r = 1...|Y'_j| \\
& \sum_i B'_{i,j} {\Theta'_{i,j}}^2 \leq \sum_r U_r^2 \ \ \forall j = 1...n', r = 1...|Y'_j| \\
& \dots \\
& \sum_i B'_{i,j} {\Theta'_{i,j}}^d \leq \sum_r U_r^d \ \ \forall j = 1...n', r = 1...|Y'_j| \\
& \sum_i B'_{i,j} / |Y'_j| \leq \alpha \ \ \forall j = 1...n'
\end{aligned}
\tag{5.2}
$$

To reduce dimensionality, we consider **Clustered Plan A**, which defines a constrained MOO problem as shown in Equation (5.2) after clustering instances and machines. After clustering, let there be m' instance clusters $X'_1, ..., X'_{m'}$ and n' machine clusters $Y'_1, ..., Y'_{n'}$, where x'_i represents the representative instance of instance cluster X'_i and y'_j represents the representative machine of machine cluster Y'_j. Let $\tilde{x}'_i$ denote the features from **Channel 1** and **Channel 2** of x'_i, and $\tilde{y}'_j$ denote the features from **Channel 4** and **Channel 5** of y'_j. Given two sets of variables B' and Θ', and assuming f is the instance-level latency prediction model, we define $f(\tilde{x}'_i, \Theta'_{i,j}, \tilde{y}'_j)$ as the latency when x'_i runs on y'_j using the resource configuration $\Theta'_{i,j}$. We assume that instances within the same instance cluster behave similarly when scheduled on machines from the same machine cluster. $\mathcal{I}(B'_{i,j})$ is an indica-

tor function of $B^{'}_{i,j}$, which indicates the number of instances in $X^{'}_{i}$ running on the machine cluster $Y^{'}_{j}$; $|X^{'}_{i}|$ denotes the number of instances in the instance cluster $X^{'}_{i}$; $|Y^{'}_{j}|$ denotes the number of machines in the machine cluster $Y^{'}_{j}$; $\sum_r U_r \in \mathbb{R}^d$ is the total capacity of the d resources on the machine cluster $Y^{'}_{j}$; α defines the maximum number of instances each machine cluster can accomodate based on diverse placement preferences, where $\alpha >= \lceil m/n \rceil$.

In the **Clustered Plan A**, the number of variables and constraints are reduced to $m' \times (d + n')$ and $m' + ((d + 1) \times n')$ respectively. These constraints ensure that the requested resources of instances allocated in the same machine cluster do not exceed its capacity.

5.2.2 Plan B: Optimization over Θ

To reduce the parameter space in the original problem, a colleague designed an IPA algorithm that takes the default configuration and determines the best machine placement strategy, B^*, that minimizes stage latency [64]. Given the optimal solution from step 1, B^*, the MORO problem in Equation (5.1) transforms to Equation (5.3).

$$\arg\min_{\Theta} \begin{bmatrix} L(B^*, \Theta) = \max_{i,j} B^*_{i,j} f(\tilde{x}_i, \Theta_i, \tilde{y}_j) \\ C(B^*, \Theta) = \sum_{i,j} B^*_{i,j} f(\tilde{x}_i, \Theta_i, \tilde{y}_j)(\mathbf{w} \cdot \Theta_i^T) \\ \dots \end{bmatrix} \tag{5.3}$$
$$s.t.\ \sum_i B^*_{i,j}\Theta^1_i \leq U^1_j,\ \dots,\ \sum_i B^*_{i,j}\Theta^d_i \leq U^d_j\ , \forall j = 1...n$$

Similar to **Plan A**, Equation (5.3) can be directly solved by existing MOO methods, where the number of variables is $m \times d$ and the number of constraints is $n \times d$. Note that both m and n can be tens of thousands, resulting in a high-dimensional problem.

To further reduce the number of variables and constraints, we consider the **Clustered Plan B**, which addresses the optimization problem after clustering instances.

After clustering, we have m' instance clusters $X'_1, ..., X'_{m'}$ and n machines (y_1, ..., y_n), where x'_i is the representative instance of instance cluster X'_i, and y_j denotes the allocated machine for x'_i after B^* is determined. Let $\tilde{x}'_i$ denote the features from **Channel 1** and **Channel 2** of the representative instance x'_i in instance cluster X'_i, and $\tilde{y}_j$ denote the features from **Channel 4** and **Channel 5** of the allocated machine y_j.

To align with the instance clustering in **Clustered Plan A**, we use B'^* to denote the instance placement in **Clustered Plan B**, where B' and B'^* differ. There are two main differences. First, instance clustering differs between **Plan A** and **Plan B**. In **Plan A**, where elements in B' are variables, instance clustering considers only instance features (e.g., cardinality). In **Plan B**, B'^* is predetermined, allowing instance clustering to consider both instance features and machine state. This results in more sub-clusters than in **Plan A**, leading to a different number of instance clusters (rows in B' and B'^*). Second, B'^* in **Plan B** is predefined, with ${B'^*}_{i,j}$ being a binary value indicating whether the representative instance of instance cluster X'_i is scheduled on specific machine y_j rather than a machine cluster. In contrast, columns of B'_j in **Plan A** represent machine clusters. Therefore, the number of columns differs between B' in **Plan A** and B'^* in **Plan B**.

In the **Clustered Plan B**, we denote the variables as Θ'. Assuming f is the instance-level latency prediction model, we define $L'_{i,j} = f(\tilde{x'}_i, \Theta'_{i,j}, \tilde{y}_j)$ as the latency when x'_i runs on y_j using the resource configuration $\Theta'_{i,j}$. We assume groups of instances may exhibit similar behavior. After instance clustering, the constrained MOO problem is defined as shown in Equation (5.4):

$$\arg\min_{\Theta'} \begin{bmatrix} L(B'^*, \Theta') = \max_{i,j} {B'^*}_{i,j} f(\tilde{x'}_i, \Theta'_{i,j}, \tilde{y}_j) \\ C(B'^*, \Theta') = \sum_{i,j} {B'^*}_{i,j} f(\tilde{x'}_i, \Theta'_{i,j}, \tilde{y}_j)(\mathbf{w} \cdot {\Theta'}_i^T)|{X'}_i| \end{bmatrix} \quad (5.4)$$
$$s.t.\ \textstyle\sum_i {\Theta'}_{i,j}{}^1 {\gamma'}_i^j \le U_j^1, ..., \sum_i {\Theta'}_{i,j}{}^d {\gamma'}_i^j \le U_j^d,\ \ \forall j = 1...n$$

where $|{X'}_i|$ denotes the number of instances in the instance cluster X'_i; ${\gamma'}_i^j$ denotes the number of instances in cluster X'_i allocated on the same machine y_j, where

$\gamma'^{j}_{i} = \sum_i B'^{*}_{i,j}, i \in I$, and I represents the indices of instances in cluster X'_i. These constraints ensure that the requested resources of instances allocated in the same machine do not exceed its capacity. In **Clustered Plan B**, the number of variables can be reduced to $m' \times d$.

Overall, **Plan A** provides a general problem statement, considering both placement plans $\boldsymbol{B}$ and resource plans $\boldsymbol{\Theta}$ as variables. **Plan B** presents a formal problem statement for optimization in RAA, with $\boldsymbol{\Theta}$ as variables under an optimal placement plan $\boldsymbol{B^*}$. We will demonstrate both scenarios in our experimental study.

This thesis focuses on the algorithm design for resource optimization in **Plan B** (RAA), where the resource configurations of instances are optimized with multiple performance goals. This optimization is performed after the IPA determines the placement plan for a stage, scheduling each instance to run on a specific machine. Additional details on IPA, provided by our colleague, can be found in our technical report [63].

5.3 Resource Assignment Advisor (RAA)

In this section, we propose a MOO algorithm design within the Resource Assignment Advisor (RAA) module—a hierarchical multi-objective optimization (MOO) approach to tune the resource plan of each instance. This aims to optimize stage latency, cloud cost, and other objectives.

5.3.1 Overview of Optimization in RAA

Since the stage-level values are aggregated from the instance-level values, the optimization problem defined in **Plan B** is reformulated as follows:

Stage-level MOO. Consider the stage-level MOO problem defined in Definition 5.1.1. 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:

$$\arg\min_{\Theta} \; F^{\Theta} = F(\Theta) = \begin{bmatrix} F_1(\Theta) = g_1(f_1(\Theta_1), ..., f_1(\Theta_m)) \\ ... \\ F_k(\Theta) = g_k(f_k(\Theta_1), ..., f_k(\Theta_m)) \end{bmatrix} \tag{5.5}$$

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`.

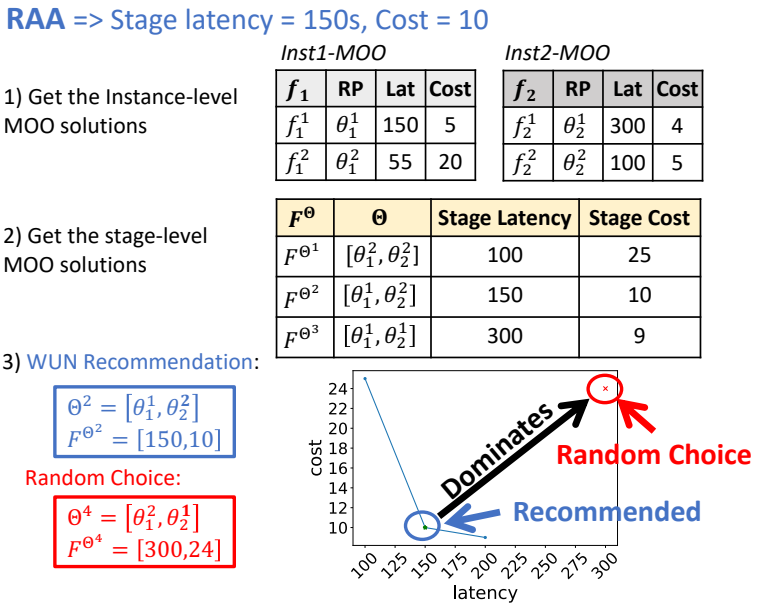
As in the instance-level, we define the **stage-level resource configuration** as $\Theta = [\theta_1^{l_1}, \theta_2^{l_2}, ...\theta_m^{l_m}]$, with its corresponding F^{Θ} in the objective space $\Phi \subseteq \mathbb{R}^k$. Then we can define stage-level **Pareto Optimality** and the **Pareto Set** similarly as shown in Section 2.1.1.

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 of stage-level objective values. In Figure 5.4, 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 Equation (5.5), it is still subject to a large number ($O(md)$) of variables. To enable a fast algorithm, we next introduce our Hierarchical MOO approach developed in the divide-and-conquer paradigm.

Definition 5.3.1. 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. Suppose 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



f_1	RP	Lat	Cost
f_1^1	θ_1^1	150	5
f_1^2	θ_1^2	55	20

f_2	RP	Lat	Cost
f_2^1	θ_2^1	300	4
f_2^2	θ_2^2	100	5

F^{Θ}	Θ	Stage Latency	Stage Cost
F^{Θ^1}	$[\theta_1^2, \theta_2^2]$	100	25
F^{Θ^2}	$[\theta_1^1, \theta_2^2]$	150	10
F^{Θ^3}	$[\theta_1^1, \theta_2^1]$	300	9

Figure 5.4: Example of RAA

denote the j-th Pareto-optimal solution in the i-th instance by using θ_i^j.

Example of RAA. In Figure 5.4, 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 $\Theta = [\theta_1^2, \theta_2^1]$ 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. Finally, we further filter solutions being dominated in the chosen set and get the stage-level MOO solutions as $[[100, 25], [150, 10], [300, 9]]$. After getting the stage-level MOO solutions, one optimal solution is recommended by Weighted Utopia Nearest (WUN) in our prior work [86].

5.3.2 General Hierarchical MOO solution

Suppose that there are k user objectives, where k_1 objectives use the `max` and k_2 objectives use `sum`, $k_1 + k_2 = k$. The high-level idea is to keep the optimality of stage-level k_1 and k_2 objective values respectively as they can be addressed differently.

For a k_1 objective, since its stage-level value is the maximum among all instance-

Algorithm 1: 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*($\boldsymbol{f}$)
3: k1Combs = *find_all_possible_values*($\boldsymbol{f}$, 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_Θ.*append*(Θ), 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

level k_1 objective values, it exists in one of the instance-level Pareto solutions. Thus it is possible to get all the stage-level k_1 objective values from the given instance-level Pareto sets. For example in Figure 5.4 with latency as k_1 objective, we can easily enumerate all the stage-level latency values as $(100, 150, 300)$. Under each stage-level latency value, it is important to efficiently make optimal decisions by considering k_2 objectives. For instance, in the same example, when stage-level latency is 300, it is expected to quickly find the stage-level solution $[300, 9]$ with $\Theta = [\theta_1^1, \theta_2^1]$ rather than $[300, 24]$ with $\Theta = [\theta_1^2, \theta_2^1]$ as the latter one is dominated. For a k_2 objective, we cannot easily get all its stage-level values as it sums up all instance-level k_2 objective values.

Algorithm 1 gives the full description of the General hierarchical MOO algorithm. 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,

Table 5.1: Additional notation for the proof of General Hierarchical MOO Problem

Symbol	Description
k_1	the number of objectives with `max` operator.
k_2	the number of objectives with `sum` operator.
f_{ijv}	the v-th objective value of j-th solution in i-th instance.
F_h	the stage-level h-th objective value in k_1 objectives ($h \in [1, ..., k_1]$).
F_v	the stage-level v-th objective value in k_2 objectives ($v \in [1, ..., k_2]$).
w_v	the weight setting of v-th objective, $v \in [1, ..., k_2]$.

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.

Proposition 5.3.1. For a stage-level MOO problem, Algorithm 1 guarantees to find a subset of stage-level Pareto optimal points.

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

Resource plan recommendation. After getting the stage-level Pareto set, we reuse UDAO's Weighted Utopia Nearest (WUN) strategy to recommend the resource plan [86], 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.

5.3.3 Theorectical Analysis

This section provides proof and complexity analysis for Algorithm 1. Table 5.1 shows the notations used in the proof of Proposition 5.3.1.

Proposition 5.3.1 For a stage-level MOO problem, Algorithm 1 guarantees to find a subset of stage-level Pareto optimal points.

Now we first prove for a special case: when there are only k_2 objectives using the `sum` operator.

Lemma 5.3.1. *For a stage-level MOO problem with k_2 objectives that use the `sum`*

operator only, Algorithm 1 guarantees to find a subset of stage-level Pareto optimal points.

In proving Lemma 5.3.1, we observe that Algorithm 1 is essentially a Weighted Sum procedure over Functions (WSF). Indeed we will prove the following two Propositions: 1) each solution returned by WSF is Pareto optimal; 2) the solution returned by the function *find_optimal* is equivalent to the solution returned by WSF. Then it follows that the solution returned by Algorithm 1 is Pareto optimal.

To introduce WSF, we first introduce the indicator function x_{ij}, $i \in [1, ..., m], j \in [1, ..., p_i]$, to indicate that the j-th solution in i-th instance is selected to contribute to the stage-level solution. p_i is the number of Pareto-optimal solutions in i-th instance. $\sum_{j=1}^{p_i} x_{ij} = 1$ means that only one solution is selected for each instance. Then $x = [x_{1j_1}, ...x_{mj_m}]$ represents the $0/1$ selection for all m instances to construct a stage-level solution.

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

$$\begin{aligned} F_v = H_v(x) = \sum_{i=1}^{m}\sum_{j=1}^{p_i} x_{ij} \times f_{ijv}, \\ where \sum_{j=1}^{p_i} x_{ij} = 1, i \in [1, ..., m], j \in [1, ..., p_i], v \in [1, ...k_2] \end{aligned} \tag{5.6}$$

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

$$\underset{x}{\text{argmin}}(\sum_{v=1}^{k_2} w_v \times H_v(x)) \tag{5.7}$$

$$\text{s.t.} \quad \sum_{v=1}^{v} w_v = 1 \tag{5.8}$$

Next, we prove for Lemma 5.3.1. As stated before, It is done in two steps.

Proposition 5.3.2. The solution constructed using x returned by WSF is Pareto optimal.

Proof.

Assume that x^* (correspoding to $[F_1^*, ..., F_{k_2}^*]$) is the solution of WSF. Suppose

that an existing solution $[F_1^{'}, ..., F_{k_2}^{'}]$ (correspoding to $x^{'}$) dominates $[F_1^*, ..., F_{k_2}^*]$. This means that $\sum_{v=1}^{k_2} w_v \times H_v(x^{'})$ is less than that of x^*.

This contradicts that x^* is the solution of WSF. So there is no $[F_1^{'}, ..., F_{k_2}^{'}]$ dominating $[F_1^*, ..., F_{k_2}^*]$. Thus, $[F_1^*, ..., F_{k_2}^*]$ is Pareto optimal. □

Proposition 5.3.3. The optimal solution returned by the function *find_ optimal* in Algorithm 1 is equivalent to the solution constructed using x returned by WSF.

Proof.

Suppose $x^{'}$ is returned by WSF. The corresponding stage-level solution is $[F_1^{'}, ..., F_{k_2}^{'}]$

$$\begin{aligned} x^{'} &= \text{argmin}(\sum_{v=1}^{k_2} w_v \times H_v(x)) \\ &= \text{argmin}(\sum_{v=1}^{k_2} w_v \times (\sum_{i=1}^{m}\sum_{j=1}^{p_i} x_{ij} \times f_{ijv})) \\ &= \text{argmin}(\sum_{i=1}^{m}(\sum_{v=1}^{k_2}\sum_{j=1}^{p_i}(w_v \times f_{ijv}) \times x_{ij})) \end{aligned} \tag{5.9}$$

For the solution $[F_1^{''}, ..., F_{k_2}^{''}]$ returned by the function *find_ optimal* in Algorithm 1, $x^{''}$ represents the corresponding selection. It is achieved by minimizing the following formula:

$$\begin{aligned} &\sum_{i=1}^{m}(WS_{ij}|j \in [1, p_i]) \\ &= \sum_{i=1}^{m}(\sum_{v=1}^{k_2} w_v \times f_{ijv}|j \in [1, p_i]) \\ &= \sum_{i=1}^{m}(\sum_{v=1}^{k_2}\sum_{j=1}^{p_i}(w_v \times f_{ijv}) \times x_{ij}) \end{aligned} \tag{5.10}$$

where $WS_{ij} = \sum_{v=1}^{k_2} w_v \times f_{ijv}$.

So, we have:

$$x^{''} = \text{argmin}(\sum_{i=1}^{m}(\sum_{v=1}^{k_2}\sum_{j=1}^{p_i}(w_v \times f_{ijv}) \times x_{ij})) \tag{5.11}$$

Since they are of the same form and achieve their minimum at the same time, we have $x' = x''$.

□

With these two propositions, we finish the proof of Lemma 5.3.1.

Finally, we prove that for the general case that also involves k_1 objectives that use the `max` aggregate operator, Proposition 5.3.1 holds as well.

Proof. Recall that Algorithm 1 enumerates all combinations of stage-level k_1 objective values. Thus, there does not exist a stage-level k_1 solution $[F_1^{'}, \ldots, F_{k_1}^{'}]$ that cannot be found by Algorithm 1. Together with Lemma 5.3.1, this completes the proof of Proposition 5.3.1.

□

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

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

5.4 Experimental Evaluation

This section presents the evaluation of our stage optimizer. Using production traces from MaxCompute, we report the end-to-end performance of our stage optimizer using a simulator of the extended MaxCompute environment (detailed in Figure 5.5) by replaying the production traces.

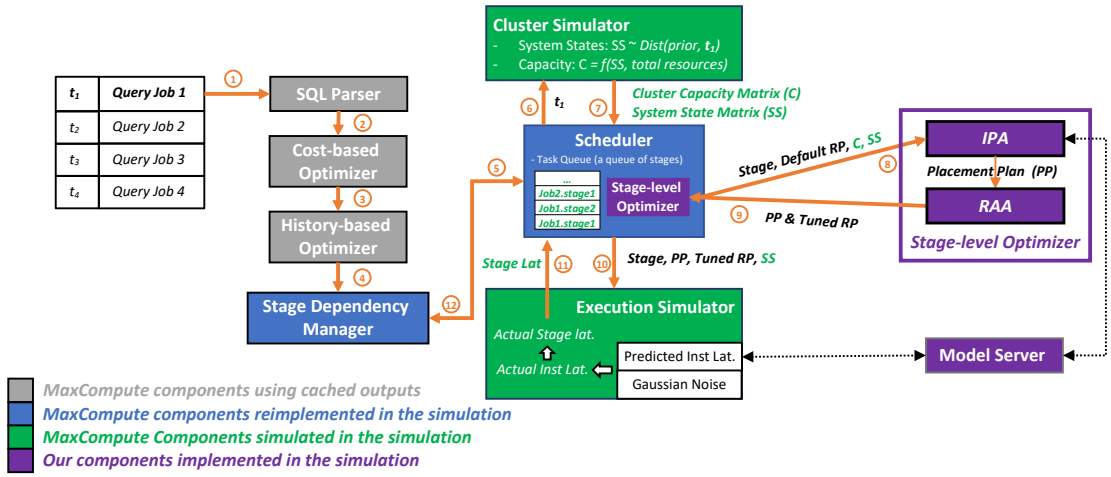


Figure 5.5: Simulation framework

5.4.1 Setup

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

Simulator. Our simulator emulates the extended MaxCompute environment depicted in Figure 5.5 [63]. First, it caches query plans and instance metadata for the workload to replace all productive traces. Second, it incorporates a stage dependency manager and an extensible scheduler capable of supporting both Fuxi and Stage-level Optimizer (SO). Third, it generates system states and capacities for each machine by sampling from prior knowledge.

To assess the performance of the proposed algorithm across various jobs, we evaluate it on three distinct production workloads.

Workload Characteristics. We analyzed three representative workloads totaling approximately 0.62 million jobs, encompassing around 2 million stages and approximately 0.12 billion instances. Each workload comprises productive jobs executed by a business department over five consecutive days at Alibaba, detailed in Table. 5.2.

Sub-workloads for RAA Evaluations. We monitored CPU utilization trends daily and averaged them across five consecutive days, using a 40-minute sliding window to identify periods of peak and minimal average CPU utilization for each workload.

Table 5.2: Workload statistics for 3 workload over 5 days

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

Subsequently, we subsampled jobs from these two time periods for each workload over the five consecutive days, resulting in 29 sub-workloads. These sub-workloads collectively include 12,000 jobs, 51,000 stages, and 6 million instances.

5.4.1.1 Resource Optimization settings

Resource Parameters. In our experiment, we focused on $d = 2$ types of resources, specifically CPU and memory. Our implementation of existing MOO methods was tailored for these two resources. However, our framework is designed to be readily extendable to cases where $d > 2$.

Clustering choices. We present various options for clustering instances in RAA:

1. `RAA(Fast_MCI)`: Implements a customized strategy based on Machine-Clustered Instance (MCI) to cluster instances. It leverages instance clusters derived from IPA's clustering result, which employs fast 1D clustering [63].

2. `RAA(DBSCAN)`: Utilizes the `DBSCAN` clustering algorithm [24] to partition instances into clusters based on their features.

3. `RAA(W/O_C)`: This approach does not involve clustering instances during algorithm execution, potentially achieving the best stage-level latency (excluding solving time) and cost. However, it typically incurs higher overhead compared to other methods.

Furthermore, when a clustering method is applied, each instance cluster is represented by a single instance, where the latency of the cluster equals that of its representative instance, and the cost is the representative instance's cost multiplied by the cluster size. After clustering, we use the term 'instance' to refer to an in-

stance cluster for simplicity, provided it does not cause confusion. By default, we choose `RAA(Fast_MCI)` for its efficiency.

MOO Baselines. We denote the solutions to the hierarchical MOO problem, obtained by applying a general hierarchical MOO solution (Algorithm 1) to address stage-level MOO problems, as `RAA(General)`. Baseline MOO algorithms are distinguished by their specific names: `WS`, `EVO`, and `PF`. To distinguish between **Plan A** and **Plan B** as defined in Section 5.2, we use `IPA` to indicate results from **Plan B**, where the placement plan is determined by `IPA(Cluster)`. For instance, results for `EVO` under **Plan A** and **Plan B** are denoted as `EVO` and `IPA+EVO`, respectively.

Implementation details for these algorithms are provided as follows.

Method 1: Evolutionary algorithm. We apply NSGA-II [22], a well-known algorithm in the family of Evolutionary (`EVO`) algorithms [23] to solve the MOO problem of both clustered **Plan A** and **Plan B**. We performed hyperparameter tuning to determine the optimal population size and the number of iterations for both **Plan A** (`EVO`) and **Plan B** (`IPA+EVO`). The NSGA-II is implemented using the Platypus library [78]. The technical details are as follows.

1. **Complex functions and constraints:** When calculating the stage-level function values, we call the instance-level predictive model. The variables are part of the input features (i.e., **Channel 3**) to obtain latency predictions. Additionally, the constraints in **Plan A** are quadratic, adding complexity compared to a linear representation.

2. **Evaluation time:** As discussed previously, the number of variables and constraints is related to the number of instances, causing the time cost of different stages to vary significantly if the stages have very different numbers of instances. Based on our observation, a stage could take over 10 minutes to return solutions, which is not acceptable given the requirement of efficiency and the fact that we have 0.1M-1M stages running. Therefore, we limit the time cost to 60 seconds for solving a stage MOO problem.

3. **Hyperparameter tuning:** Population size and the number of iterations are two crucial hyperparameters influencing optimization performance. To select

an optimal set of hyperparameters within the 60-second time limit, we test 12 sets of hyperparameters on a workload consisting of 103 stages, where the number of instances varies from 1 to over 6,000. The final hyperparameters are selected based on the highest coverage of feasible stages and the highest reduction rate in stage-level latency and cost.

4. **Randomness:** NSGA-II is an evolutionary algorithm that initializes the population randomly and executes genetic operations (such as crossover and mutation) based on a given probability. To ensure reproducibility of the results, we fix the randomness for each iteration related to population generation and genetic operations.

Method 2: Weighted Sum. We applied `WS(Sample)` [69] to reduce a MOO problem into a single-objective optimization (SOO) problem by minimizing the weighted sum of the stage latency and cost. In our implementation, we used a random sampling approach to solve the SOO problem. For each stage, we obtained the corresponding MOO solutions over a set of weights for latency and cost by the following steps:

1. Sample up to 100,000 variable choices randomly.
2. Filter out variables that violate resource capacity constraints or diverse preferences.
3. Evaluate the objective values for each filtered variable.
4. Iterate the objective weights for latency and cost and return the variables that minimize the weighted sum of the objectives.
5. Construct the MOO solutions based on the returned variables.

Method 3: Progressive Frontier (PF). We implemented our previous work `PF(MOGD)` in a parallel version (PF-AP) to progressively solve the MOO problem [86]. `PF(MOGD)` transforms the MOO problem into a series of single-objective constrained optimization (CO) problems and employs the Multi-Objective Gradient Descent (MOGD) solver to solve them.

5.4.2 Resource Optimization (RO) Evaluation

This section presents the results from the aspects of clustering methods, the baseline MOO algorithms, and our proposed methods for resource optimization in the *Stage-level Optimizer*.

Evaluation results. We next evaluate the end-to-end performance of our Stage-level Optimizer (SO) against the current HBO and Fuxi scheduler [115], 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 (as shown in Figure 5.5) and replayed the query traces to conduct our experiments.

We consider the following metrics in resource optimization (RO):

1. $Lat_s^{(in)}$, the average stage latency that includes the RO time;
2. $Cost_s$, the average cloud cost of all stages in a workload;
3. T_s, the Resource Optimization (RO) time cost.
4. *Coverage*, the ratio of stages that receive feasible solutions within 60s;

We use the reduction rate of latency and cost based on results from the HBO and Fuxi scheduler (i.e. denoted as default solutions). For example, if the reduction rates of the latency and cost are high, it indicates the obtained latency and cost are lower and better than the default latency and cost.

Expt 1: *IPA+RAA*. We run RAA with three choices using `IPA(Cluster)`. Referring to results from the *IPA only*, 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 [64]. As shown in Table 5.3, `IPA+RAA(W/O_C)` does RAA without clustering and suffers high overhead (up to 19s for a stage). `IPA+RAA(DBSCAN)` applies DBSCAN [24] 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 incurs up to 241 msec. Compared to *IPA only*, all the choices of RAA reduce more stage

Table 5.3: Average Reduction Rate (RR) against Fuxi in 29 sub-workloads within 60s

	Coverage (%)			$Lat_s^{(in)} \downarrow$ (%)			$Cost_s \downarrow$ (%)			$avg(T_s)$ (ms) / $\max(T_s)$ (ms)		
SO choice	A	B	C	A	B	C	A	B	C	A	B	C
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**
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

latency (including the solving time) and cost, where IPA+RAA(General) performs the best which reduces stage latency by 36-80% and cost by 29-75%.

Expt 2: *MOO baselines.* We next compare to SOTA MOO solutions, EVO [23], WS(Sample) [69], and PF(MOGD) [86], using Definition 5.1.1. As shown in the 3 red rows of Table 5.3, (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(General), 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 Equation (5.5). As shown in the last 3 blue rows in Table 5.3, they are still inferior to IPA+RAA(General) in both latency and cost reduction and in running time (mostly taking 1-6 sec to complete).

5.4.3 Breakdown Analysis

This section provides a detailed analysis of IPA+RAA and demonstrates the adaptivity of our resource optimization approach.

Analysis on IPA+RAA of three workloads. Figures 5.6-5.8 illustrate the distribution of reduction rates in stage latency (both including and excluding solving time) as well as total cost for three distinct workloads, with IPA(Cluster) as the default choice.

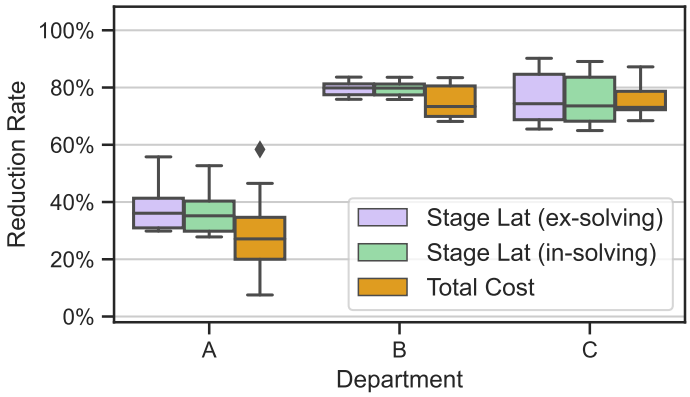


Figure 5.6: RAA with instance clustering (General)

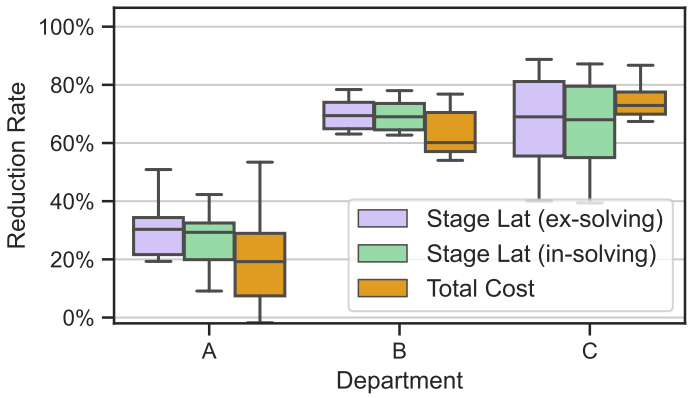


Figure 5.7: RAA with instance clustering (DBSCAN)

For the workload with the most long-running jobs (workload C), our approach `IPA+RAA(General)` outperforms `IPA+RAA(DBSCAN)` across all workloads. Specifically, compared to `IPA+RAA(W/O_C)`, `IPA+RAA(General)` achieves a significantly higher latency reduction rate (including solving time), reaching at least 60% compared to approximately 40%. For the workloads predominantly composed of short-running jobs (workloads A and B), `IPA+RAA(General)` shows a comparable performance in stage latency (excluding solving time) and cost reduction rate to that of `IPA+RAA(W/O_C)`.

Analysis on adaptivity of resource optimization. The key insight of Multi-Objective Resource Optimization (MORO) within a stage is the adaptive allocation of resources to instances. For example, it allocates more resources to long-running instances while assigning fewer resources to short-running instances.

Figure 5.9 demonstrates the performance of a long-running stage comprising 479 instances with similar input cardinality, focusing on their latency, cost, cardinality,

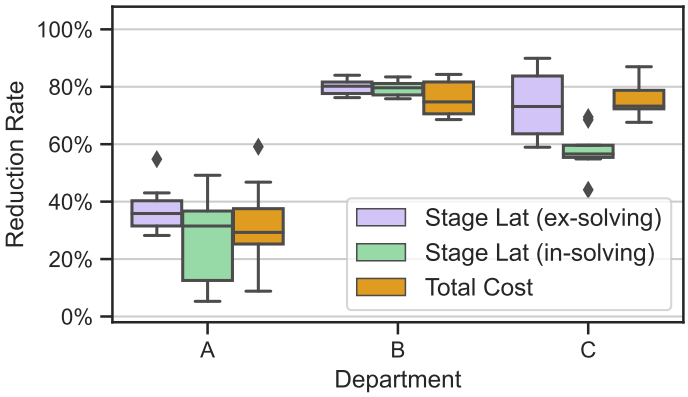
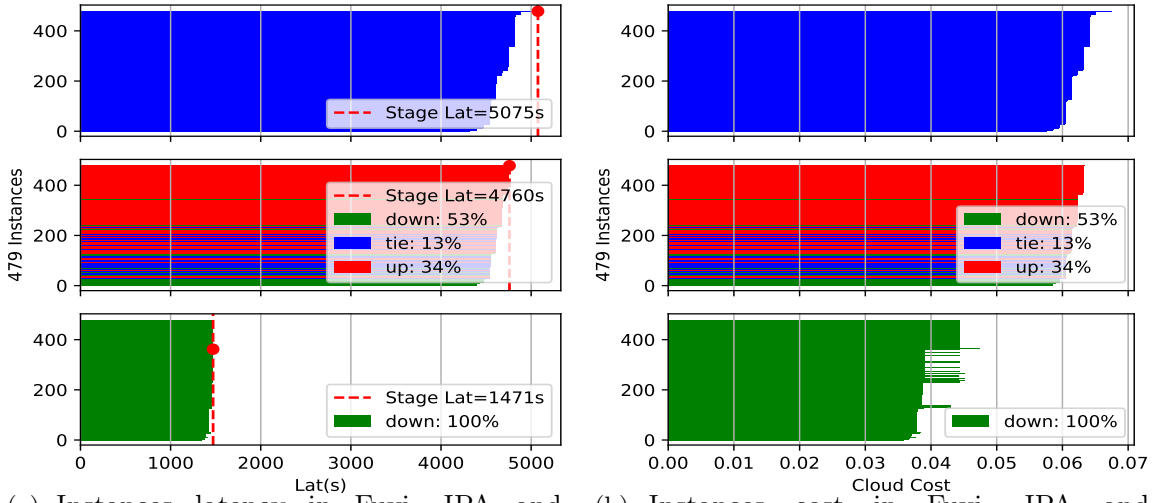


Figure 5.8: RAA without instance clustering



(a) Instances latency in Fuxi, IPA and IPA+RAA

(b) Instances cost in Fuxi, IPA and IPA+RAA

Figure 5.9: An example of stage latency and cost analysis

and resource plans. Figure 5.9(a) and 5.9(b) show the latency and cost of each instance by using Fuxi (row 1), `IPA` only (row 2), and `IPA+RAA` (row3). IPA reduces stage latency by decreasing latency for 53% of instances and increasing it for 34% of instances. When combined with `RAA`, `IPA+RAA` further reduces latency and cost for all instances by optimizing the resource plan. In this example, the corresponding adjusted resource plans are shown in Figure 5.10. `IPA+RAA` adjusts CPU core assignments and increases memory size to 6x for each instance, outperforming the Fuxi policy in both latency and cost.

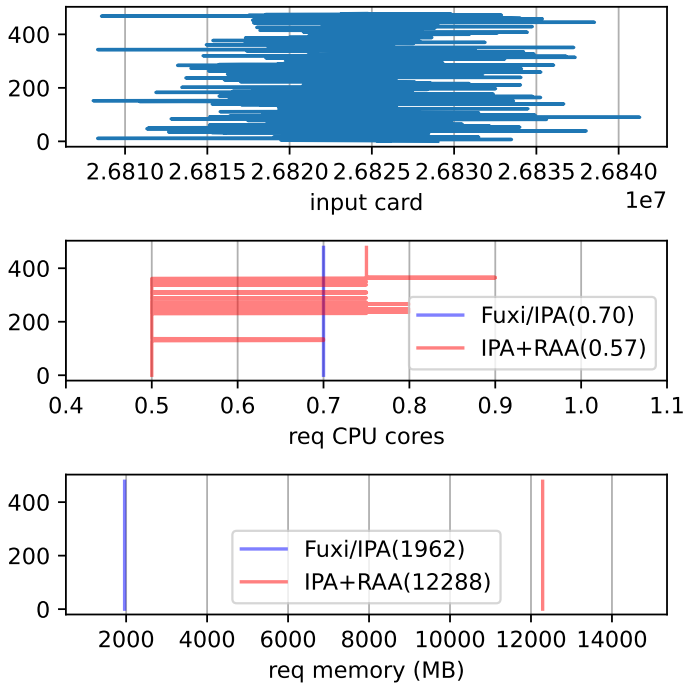


Figure 5.10: Instance cardinality and resource plans in Fuxi, IPA and IPA+RAA

5.5 Summary

We presented an algorithm design for multi-objective resource optimization based on the MaxCompute [71] big data system. To address the complexity of our system, our *Stage-level Optimizer* exploits fine-grained instance-level models and develops a novel `RAA` module to derive instance-specific resource plans, thereby reducing stage latency and cost within a hierarchical MOO framework. Evaluation using production workloads shows that `IPA+RAA` achieved the reduction of 36-80% latency and 29-75% cost while running in 0.02-0.24s compared to the Fuxi scheduler [115]. It could reduce 4-77% latency and up to 48% cost at the same time, compared to the existing MOO methods.

CHAPTER 6

Query-level Optimizer

The previous chapter aimed at stage-level Pareto optimality, while this chapter targets query-level Pareto optimality. In addition, it is motivated by the latest big data systems like Spark that allow fine-grained parameter tuning. For example, some parameters can be tuned independently for each query stage, while others are shared across all stages, introducing a high-dimensional parameter space and complex constraints. To address this challenge, we propose a new approach called Hierarchical MOO with Constraints (HMOOC). This method decomposes the optimization problem of a large parameter space into smaller subproblems, each constrained to use the same shared parameters. Given that these subproblems are not independent, we develop techniques to generate a sufficiently large set of candidate solutions and efficiently aggregate them to form global Pareto optimal solutions. Evaluation results using TPC-H and TPC-DS benchmarks demonstrate that HMOOC outperforms existing MOO methods, achieving a 4.7% to 54.1% improvement in hypervolume and an 81% to 98.3% reduction in solving time.

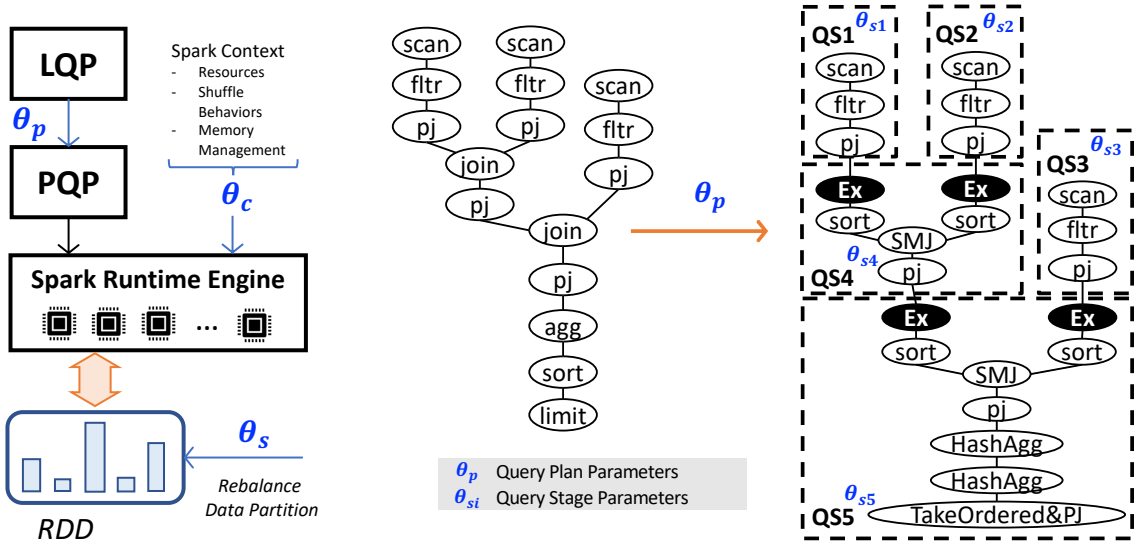
6.1 Problem Statement and Overview

In this section, we provide background on Spark including its adaptive query execution extension and present initial results illustrating the benefits and complexity of fine-grained tuning. We then formally define our Spark parameter tuning problem and provide an overview of our compile-time/runtime optimization approach.

6.1.1 Background on Spark

Apache Spark [109] is an open-source distributed computing system for large-scale data processing and analytics. The core concepts of Spark include *jobs*, representing computations initiated by actions, and *stages*, which are organized based on shuffle dependencies, serving as boundaries that partition the computation graph of a job. Stages comprise sets of *tasks* executed in parallel, each processing a specific *data partition*. *Executors*, acting as worker processes, execute these tasks on individual cluster nodes.

Spark SQL seamlessly integrates relational processing into the Spark framework [3]. A submitted SQL query undergoes parsing, analysis, and optimization to form a *logical query plan* (LQP). In subsequent physical planning, Spark transforms the LQP to one or more *physical query plans* (PQP), using physical operators provided by the Spark execution engine. Then it selects one PQP using a cost model, which mostly applies to join algorithms. The physical planner also performs rule-based optimizations, such as pipelining projections or filters into one `map` operation. The PQP is then divided into a directed acyclic graph (DAG) of *query stages* (QSs) based on data exchange dependencies such as shuffling or broadcasting. These query stages are then executed in a topological order.



(a) Mixed control in query lifetime

(b) Logical query plan (LQP) and physical query plan (PQP) of TPCH-Q3

Figure 6.1: Spark parameters provide mixed control through query compilation and execution

Table 6.1: Spark parameters in three categories

$\boldsymbol{\theta}_c$	**Context Parameters**	**Default**
k_1	`spark.executor.cores`	1
k_2	`spark.executor.memory`	1G
k_3	`spark.executor.instances`	-
k_4	`spark.default.parallelism`	-
k_5	`spark.reducer.maxSizeInFlight`	48M
k_6	`spark.shuffle.sort.bypassMergeThreshold`	200
k_7	`spark.shuffle.compress`	True
k_8	`spark.memory.fraction`	0.6
$\boldsymbol{\theta}_p$	**Logical Query Plan Parameters**	**Default**
s_1	`spark.sql.adaptive.advisoryPartitionSizeInBytes`	64M
s_2	`spark.sql.adaptive.nonEmptyPartitionRatioForBroadcastJoin`	0.2
s_3	`spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold`	0b
s_4	`spark.sql.adaptive.autoBroadcastJoinThreshold`	10M
s_5	`spark.sql.shuffle.partitions`	200
s_6	`spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes`	256M
s_7	`spark.sql.adaptive.skewJoin.skewedPartitionFactor`	5.0
s_8	`spark.sql.files.maxPartitionBytes`	128M
s_9	`spark.sql.files.openCostInBytes`	4M
$\boldsymbol{\theta}_s$	**Query Stage Parameters**	**Default**
s_{10}	`spark.sql.adaptive.rebalancePartitionsSmallPartitionFactor`	0.2
s_{11}	`spark.sql.adaptive.coalescePartitions.minPartitionSize`	1M

The execution of a Spark SQL query is configured by three categories of parameters, as shown in Table 6.1, providing different controls in query lifetime. As Figure 6.1(a) shows [65], **query plan parameters $\boldsymbol{\theta}_p$** guide the translation from a logical query plan to a physical query plan, influencing the decisions such as the bucket size for file reading and the join algorithms to use via Spark's parametric optimization rules. Figure 6.1(b) shows a concrete example of translating a LQP to PQP [65], where each logical operator is instantiated by specific algorithms (e.g., the first join is implemented by sorting both input relations and then a merge join of them), additional exchange operators are injected to realize data exchanges, and query stages are identified at the boundaries of exchange operators. Further, **query stage parameters $\boldsymbol{\theta}_s$** control the optimization of a query stage via parametric rules, such as rebalancing data partitions. Finally, **context parameters $\boldsymbol{\theta}_c$**, specified on the Spark context, control shared resources, shuffle behaviors, and memory manage-

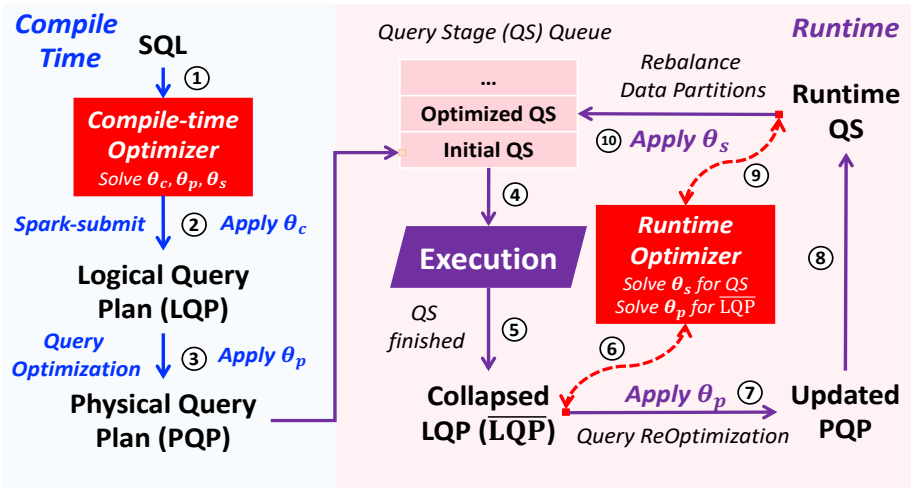
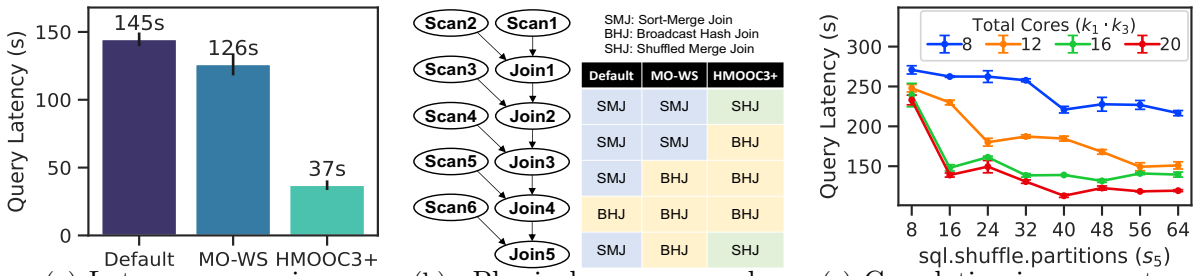


Figure 6.2: Query life cycle with an optimizer for parameter tuning

ment throughout query execution. While they are in effect only at runtime, $\boldsymbol{\theta}_c$ must be specified at the query submission time when the Spark context is initialized.

Adaptive Query Execution (AQE). Cardinality estimation [32, 59, 62, 75, 79, 88, 101, 105, 106, 107, 117] has been a long-standing issue that impacts the effectiveness of the physical query plan. To address this issue, Spark introduced *Adaptive Query Execution* (AQE) that enables runtime optimization based on precise statistics collected from completed stages [25]. Figure 6.2 shows the life cycle of a SQL query with the AQE mechanism turned on [65]. At compile time, a query is transformed to a LQP and then a PQP through query optimization (step 3). Query stages (QSs) that have their dependencies cleared are then submitted for execution. During query runtime, Spark iteratively updates LQP by collapsing completed QSs into dummy operators with observed cardinalities, leading to a so-called collapsed query plan $\overline{\text{LQP}}$ (step 5), and re-optimizes the $\overline{\text{LQP}}$ (step 7) and the QSs (step 10), until all QSs are completed. At the core of AQE are runtime optimization rules. Each rule internally traverses the query operators and takes effect on them. These rules are categorized as parametric and non-parametric, and each parametric rule is configured by a subset of $\boldsymbol{\theta}_p$ or $\boldsymbol{\theta}_s$ parameters. The details of those rules are in our tech report [65].



(a) Latency comparison (b) Physical query plan choices (c) Correlation in parameters

Figure 6.3: Profiling TPCH-Q9 (12 subQs) over different configurations

6.1.2 Effects of Parameter Tuning

We next consider the issue of Spark parameter tuning and present initial observations that motivated our approach.

First, *parameter tuning affects performance.* While Spark supports AQE through parametric and non-parametric rules, it does not support parameter tuning itself. The first observation that motivated our work is that tuning over a mixed parameter space is crucial for Spark performance. Figure 6.3(a) shows that for TPCH-Q9, query-level parameter tuning using a prior MOO method [86] and then running AQE (the middle bar) can already provide a 13% improvement over AQE with the default configuration (left bar) [65].

Second, *fine-grained tuning has performance benefits over query-level tuning.* While existing work on Spark parameter tuning [51, 53, 55, 56, 86, 108] focuses on query-level tuning, we show in Figure 6.3(a) that adapting $\boldsymbol{\theta}_p$ for different collapsed query plans during runtime can further reduce the latency by 61% (the right bar) [65]. Figure 6.3(b) shows the simplified query structure of TPCH-Q9, including 6 scan operators and 5 join operators [65]. Adapting $\boldsymbol{\theta}_p$ for different collapsed query plans with observed statistics allows us to discover a new physical query plan with 3 broadcast hash joins (BHJs) and 2 shuffled hash joins (SHJs), outperforming the query-level tuning result with 2 sort-merge joins (SMJs) + 3 BHJs.

Third, *the parameters that are best tuned at runtime based on precise statistics are correlated with the parameters that must be set at submission time.* While the $\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}_s$ parameters are best tuned at runtime to benefit from precise statistics, they are strongly correlated with the Spark context parameters, $\boldsymbol{\theta}_c$, which control

shared resources and must be set at query submission time when the Spark context is initialized. Figure 6.3(c) illustrates that the optimal choice of s_5 in $\boldsymbol{\theta}_p$ is strongly correlated with the total number of cores $k_1 * k_3$ configured in $\boldsymbol{\theta}_c$ [65]. Many similar examples exist.

6.1.3 Our Parameter Tuning Approach

We next introduce our approach that supports multi-granularity parameter tuning using hybrid compile-time/runtime optimization and formally define the optimization problem in the MOO setting.

6.1.3.1 Hybrid, Multi-Granularity Tuning

The goal of our work is to find, for each Spark query, the optimal configuration of all the $\boldsymbol{\theta}_p$, $\boldsymbol{\theta}_s$, and $\boldsymbol{\theta}_c$ parameters under *multi-granularity tuning*. While the context parameters $\boldsymbol{\theta}_c$ configure the Spark context at the *query level*, we tune other parameters at fine granularity to maximize performance gains, including setting the query plan parameters $\boldsymbol{\theta}_p$ distinctly for each *collapsed query plan* and the query stage parameters $\boldsymbol{\theta}_s$ for each *query stage* in the physical plan.

To address the correlation between the context parameters $\boldsymbol{\theta}_c$ (set at query submission time) and $\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}_s$ parameters (best tuned at runtime), we introduce a *hybrid compile-time / runtime optimization* approach, as depicted by the two red boxes in Figure 6.2 [65].

Compile-time: Our goal is to (approximately) derive the optimal $\boldsymbol{\theta}_c^*$, by leveraging the correlation of all the parameters, to construct an ideal Spark context for query execution. Our compile-time optimization uses cardinality estimates by Spark's cost optimizer.

Runtime: With Spark context fixed, our runtime optimization runs as a plugin of AQE, invoked each time the collapsed query plan ($\overline{\text{LQP}}$) is updated from a completed query stage, and adjusts $\boldsymbol{\theta}_p$ for the collapsed query plan based on the latest runtime statistics. Then AQE applies $\boldsymbol{\theta}_p$ to its parametric rules to generate an updated physical query plan ($\overline{\text{PQP}}$). For the query stages in this new physical plan, our runtime optimization kicks in to optimize $\boldsymbol{\theta}_s$ parameters based on precise statistics.

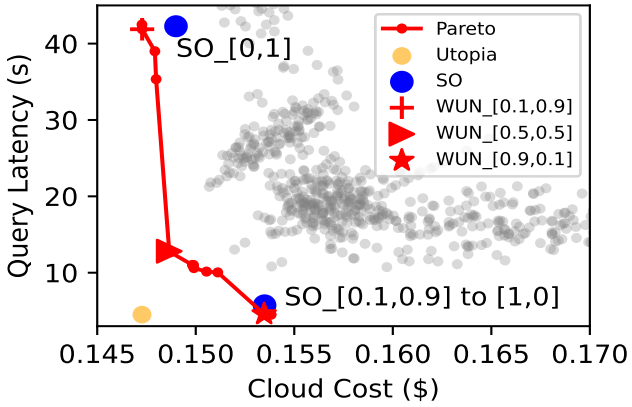


Figure 6.4: MOO solutions for TPCH Q2

Then AQE applies parametric rules with the tuned $\boldsymbol{\theta}_s$ to optimize data partitions of these stages.

This thesis focuses on *algorithm design for compile-time optimization*, emphasizing its ability to address parameter diversity and complexity of fine-grained parameter control at compile-time rather than at runtime.

6.1.3.2 Multi-Objective Optimization

Targeting cloud use, our optimization problem concerns multiple user objectives such as query latency and cloud cost in terms of CPU hours or a weighted combination of CPU, memory, and IO resources.

Recall the explanation in Section 3.3 that solving MOO problems is not the same as using fixed weights to create a single-objective (SO) optimization problem. Figure 6.4 illustrates this for TPCH-Q2 in the 2D space of query latency and cloud cost: 11 SO problems generated from evenly spaced weight vectors return only two distinct solutions (marked by the blue dots), where 10 of them collide to the same bottom point. Increasing to 101 weight vectors still returns only 3 distinct points.

Figure 6.4 shows a Pareto front for TPCH-Q2. Most configurations (e.g. grey dots) are dominated by the Pareto optimal configurations (e.g. red dots) in both objectives. Hence, the MOO solution allows us to skip the vast set of dominated configurations. Furthermore, the Pareto points themselves represent tradeoffs between the two competing objectives. The optimizer can recommend one of them based on the user preference, e.g., favoring latency to cost in peak hours with weights 0.9 to

0.1 and vice-versa in off-peak hours. The recommendation can be made based on the Weighted Utopia Nearest (WUN) distance [86] of the Pareto points from the Utopia point $\boldsymbol{U}$, which is the hypothetical optimum in all objectives, marked by the orange dot in the figure. Figure 6.4 illustrates several recommendations generated by running WUN on the Pareto front with varying weighted preferences for the objectives.

We next define the MOO problem for Spark parameter tuning.

Definition 6.1.1. Multi-Objective Optimization for Spark SQL

$$\arg \min_{\boldsymbol{\theta_c}, \{\boldsymbol{\theta_p}\}, \{\boldsymbol{\theta_s}\}} \boldsymbol{f}(\boldsymbol{\theta_c}, \{\boldsymbol{\theta_p}\}, \{\boldsymbol{\theta_s}\}) = \begin{bmatrix} f_1(\text{LQP}, \boldsymbol{\theta_c}, \{\boldsymbol{\theta_p}\}, \{\boldsymbol{\theta_s}\}, \alpha, \beta, \gamma) \\ ... \\ f_k(\text{LQP}, \boldsymbol{\theta_c}, \{\boldsymbol{\theta_p}\}, \{\boldsymbol{\theta_s}\}, \alpha, \beta, \gamma) \end{bmatrix} \tag{6.1}$$

$$\begin{aligned} & \boldsymbol{\theta_c} \in \Sigma_c, \\ s.t. \quad & \{\boldsymbol{\theta_p}\} = \{\boldsymbol{\theta}_{p1}, \boldsymbol{\theta}_{p2}, ..., \boldsymbol{\theta}_{pt}, ...\}, \forall \boldsymbol{\theta}_{pt} \in \Sigma_p \\ & \{\boldsymbol{\theta_s}\} = \{\boldsymbol{\theta}_{s1}, \boldsymbol{\theta}_{s2}, ..., \boldsymbol{\theta}_{si}, ...\}, \forall \boldsymbol{\theta}_{si} \in \Sigma_s \end{aligned}$$

where LQP denotes the logical query plan with operator cardinality estimates, and $\boldsymbol{\theta_c}, \{\boldsymbol{\theta_p}\}, \{\boldsymbol{\theta_s}\}$ represent the *decision variables* configuring Spark context, LQP transformations, and query stage (QS) optimizations, respectively. More specifically, $\{\boldsymbol{\theta_p}\}$ is the collection of all LQP parameters, and $\boldsymbol{\theta}_{pt}$ is a copy of $\boldsymbol{\theta}_p$ for the t-th transformation of the collapsed query plan $\overline{\text{LQP}}$. Similarly, $\{\boldsymbol{\theta_s}\}$ is the collection of QS parameters, and $\boldsymbol{\theta}_{si}$ is a copy for optimizing query stage i. $\Sigma_c, \Sigma_p, \Sigma_s$ are the feasible space for $\boldsymbol{\theta}_c$, $\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}_s$, respectively. Finally, α, β, γ are the *non-decision variables* (not tunable, but crucial factors that affect model performance), representing the input characteristics, the distribution of partition sizes for data exchange, and resource contention status during runtime.

6.2 Compile-time Optimization

In this section, we will present our compile-time optimization approach to multi-granularity parameter tuning with the multi-objective optimization setting.

6.2.1 Hierarchical MOO with Constraints

To provide a technical context for our problem, an overview of the learned models is presented, provided by our colleague to facilitate optimization at compile-time.

Compile-time model. At compile time, Spark provides a LQP and then a PQP, but no any data structures that would suit our goal of fine-grained tuning. Therefore, we introduce the notion of *compile-time subquery* (subQ) to denote a group of logical operators that will correspond to a query stage (QS) when the logical plan is translated to a physical plan. In other words, it is a sub-structure of the logical plan that is reversely mapped from a query stage in a physical plan. Figure 6.1(b) illustrates the LQP of TPCH-Q3, which can be divided into five subQs, each corresponding to one QS. As an enhancement, our work can sample multiple physical plans for each query at compile time, which will lead to different subQ structures. We collect all of these subQ structures, and develop a predictive model for each subQ to enable fine-grained tuning of $\boldsymbol{\theta}_p$ at compile-time.

Our compile-time optimization finds the optimal configuration $\boldsymbol{\theta}_c^*$ of the context parameters to construct an ideal Spark context for query execution. Our approach does so by exploring the correlation of $\boldsymbol{\theta}_c$ with fine-grained $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ parameters for different subqueries, under the modeling constraint that the non-decision variables for cardinality estimates are based on Spark's cost-based optimizer. Nevertheless, even under the modeling constraint, capturing the correlation between the mixed parameter space allows us to find a better Spark context for query execution, as we will demonstrate in our experimental study.

The multi-objective optimization problem in Definition 6.1.1 provides fine-grained control of $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$, at the subquery (subQ) level and query stage level, respectively, besides the query level control of $\boldsymbol{\theta_c}$. As such, the dimensionality of the parameter space is $d_c + m \cdot (d_p + d_s)$, where d_c, d_p and d_s denote the dimensionality of the $\boldsymbol{\theta_c}$, $\boldsymbol{\theta_p}$, $\boldsymbol{\theta_s}$ parameters, respectively, and m is the number of query stages. Such high dimensionality defeats most existing MOO methods when the solving time must be kept under the constraint of 1-2 seconds for cloud use, as we shall demonstrate in our experimental study.

To combat the high-dimensionality of the parameter space, we propose a new

approach named *Hierarchical MOO with Constraints* (HMOOC). In a nutshell, it follows a divide-and-conquer framework to break a large optimization problem on ($\boldsymbol{\theta_c}$, $\{\boldsymbol{\theta_p}\}$, $\{\boldsymbol{\theta_s}\}$) to a set of smaller problems on ($\boldsymbol{\theta_c}$, $\boldsymbol{\theta_p}$, $\boldsymbol{\theta_s}$), each corresponding to a subQ of the logical query plan. However, these smaller problems are not independent as they must obey the constraint that all the subproblems must choose the same $\boldsymbol{\theta_c}$ value. More specifically, the problem for HMOOC is defined as follows:

Definition 6.2.1. Hierarchical MOO with Constraints (HMOOC)

$$\arg\min_{\boldsymbol{\theta}} \boldsymbol{f}(\boldsymbol{\theta}) = \begin{bmatrix} f_1(\boldsymbol{\theta}) = \Lambda(\phi_1(\text{LQP}_1, \boldsymbol{\theta_c}, \boldsymbol{\theta_{p1}}, \boldsymbol{\theta_{s1}}), \ldots, \phi_1(\text{LQP}_m, \boldsymbol{\theta_c}, \boldsymbol{\theta_{pm}}, \boldsymbol{\theta_{sm}})) \\ \vdots \\ f_k(\boldsymbol{\theta}) = \Lambda(\phi_k(LQP_1, \boldsymbol{\theta_c}, \boldsymbol{\theta_{p1}}, \boldsymbol{\theta_{s1}}), \ldots, \phi_k(LQP_m, \boldsymbol{\theta_c}, \boldsymbol{\theta_{pm}}, \boldsymbol{\theta_{sm}})) \end{bmatrix} \quad (6.2)$$
$$s.t. \qquad \boldsymbol{\theta_c} \in \Sigma_c \subseteq \mathbb{R}^{d_c}, \quad \boldsymbol{\theta_{pi}} \in \Sigma_p \subseteq \mathbb{R}^{d_p}, \boldsymbol{\theta_{si}} \in \Sigma_s \subseteq \mathbb{R}^{d_s}, \quad i = 1, \ldots, m$$

where LQP_i denotes the i-th subQ of the logical plan query, $\boldsymbol{\theta_i} = (\boldsymbol{\theta_c}, \boldsymbol{\theta_{pi}}, \boldsymbol{\theta_{si}})$ denotes its configuration, with $i = 1, \ldots, m$, and m is the number of subQs. Most notably, all the subQs share the same $\boldsymbol{\theta_c}$, but can use different values of $\boldsymbol{\theta_{pi}}$ and $\boldsymbol{\theta_{ps}}$. Additionally, ϕ_j is the subQ predictive model of the j-th objective, where $j = 1, \ldots, k$. The function Λ is the mapping from subQ-level objective values to query-level objective values, which can be aggregated using `sum` based on our choice of analytical latency and cost metrics.

The main idea behind our approach is to tune each subQ independently under the constraint that $\boldsymbol{\theta_c}$ is identical among all subQ's. By doing so, we aim to get the local subQ-level solutions, and then recover the query-level Pareto optimal solutions by composing these local solutions efficiently. In brief, it includes three sequential steps: (1) **subQ tuning**, (2) **DAG aggregation**, and (3) **WUN recommendation**.

Figure 6.5 illustrates an example of compile-time optimization for TPCH-Q3 under the latency and cost objectives. For simplicity, we show only the first three subQ's in this query and omit $\boldsymbol{\theta_s}$ in this example. In subQ-tuning, we obtain subQ-level solutions with configurations of $\boldsymbol{\theta_c}$ and $\boldsymbol{\theta_p}$, where $\boldsymbol{\theta_c}$ has the same set of two values ($\boldsymbol{\theta_c^1}$, $\boldsymbol{\theta_c^2}$) among all subQ's, but $\boldsymbol{\theta_p}$ values vary. Subsequently in the DAG aggregation step, the query-level latency and cost are computed as the sum of the

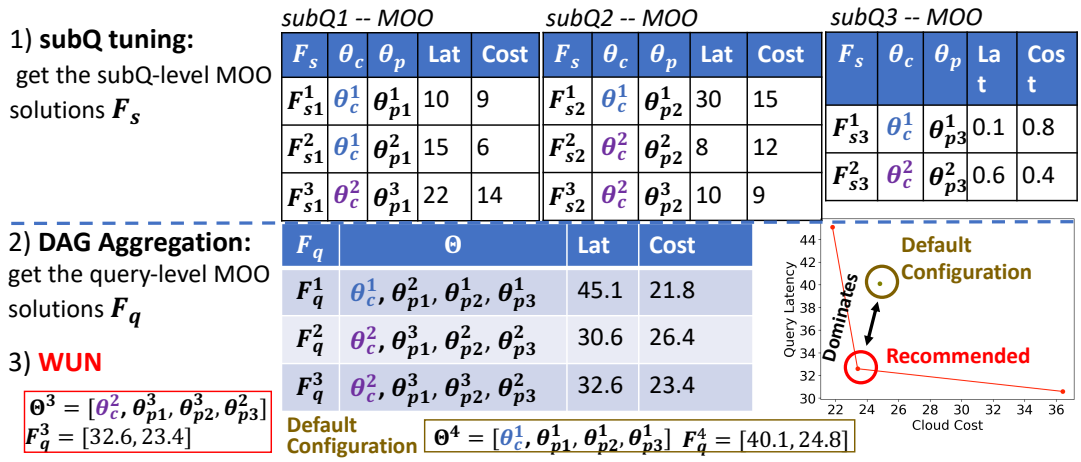


Figure 6.5: Example of the Compile-time optimization of TPCH Q3

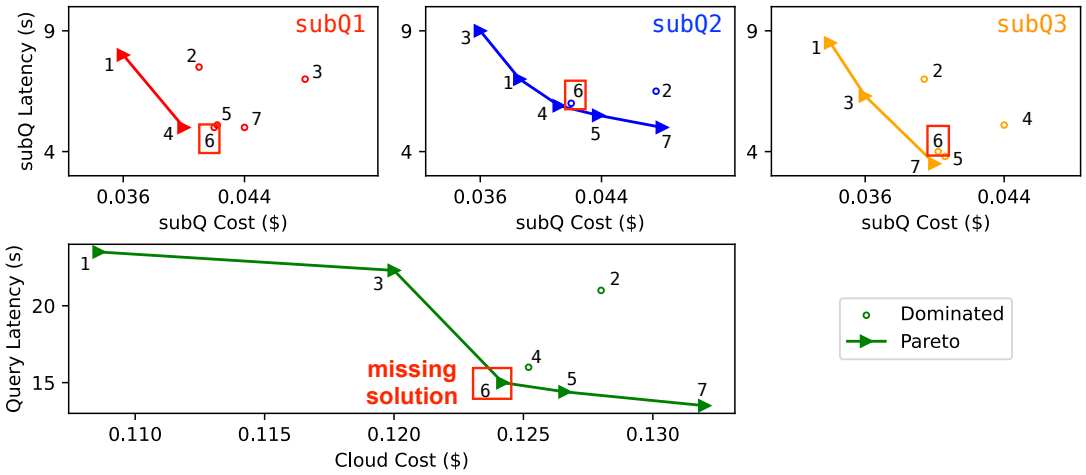


Figure 6.6: Example of missed global optimal solutions of TPCH Q3

three subQ-level latency and cost values, and only the Pareto optimal values of latency and cost are retained. Finally, in the third step, we use the WUN (weighted Utopia nearest) policy [86] to recommend a configuration from the Pareto front.

6.2.2 Subquery (subQ) Tuning

Subquery (subQ) tuning aims to generate an effective set of local solutions of ($\boldsymbol{\theta_c}$, $\boldsymbol{\theta_p}$, $\boldsymbol{\theta_s}$) for each subQ while obeying the constraint that all the subQs share the same $\boldsymbol{\theta_c}$. For simplicity, we focus on ($\boldsymbol{\theta_c}$, $\boldsymbol{\theta_p}$) in the following discussion as $\boldsymbol{\theta_s}$ is treated the same as $\boldsymbol{\theta_p}$.

One may wonder whether it is sufficient to generate only the local Pareto solutions of ($\boldsymbol{\theta_c}$, $\boldsymbol{\theta_p}$) of each subQ. Unfortunately, this will lead to missed global Pareto optimal solutions due to the constraint on $\boldsymbol{\theta_c}$. Figure 6.6 illustrates an example with

3 subQs, where solutions sharing the same index fall under the same $\boldsymbol{\theta_c}$ configuration and have achieved optimal $\boldsymbol{\theta_p}$ under that $\boldsymbol{\theta_c}$ value. The first row in Figure 6.6 showcases subQ-level solutions, where triangle points represent subQ-level optima and circle points denote dominated solutions. The second row in Figure 6.6 displays the corresponding query-level values, where both query-level latency and cost are the sums of subQ-level latency and cost. Notably, solution 6 is absent from the local subQ-level Pareto optimal solutions across all subQs. Due to the identical $\boldsymbol{\theta_c}$ constraint and the sum aggregation from subQ-level values to query-level values, although solution 6 is dominated in all subQs, the sum of its subQ-level latency and cost performs better than solution 4 (constructed from subQ-level Pareto optimal solutions).

Our main idea is to maintain an effective set of solutions, more than just local Pareto solutions, for each subQ in order to recover query-level Pareto optimal solutions. To do so, we introduce the following two techniques.

1. Enriching $\boldsymbol{\theta_c}$ Candidates. To minimize the chance of missing global solutions, we seek to construct a diverse, effective set of $\boldsymbol{\theta_c}$ configurations to be considered across all subQs. $\boldsymbol{\theta_c}$ can be initialized by random sampling or grid-search over its domain of values. Then, we enrich the $\boldsymbol{\theta_c}$ set using a few methods. Drawing inspiration from the evolutionary algorithms [23], we introduce a *crossover* operation over the existing $\boldsymbol{\theta_c}$ population to generate new candidates. If crossover cannot generate more candidates, e.g., for some grid search methods used for initial sampling of $\boldsymbol{\theta_c}$ candidates, then we add random sampling to discover new candidates.

2. Optimal $\boldsymbol{\theta_p}$ Approximation. Next, under each $\boldsymbol{\theta_c}$ candidate, we show that it is crucial to keep track of the local Pareto optimal $\boldsymbol{\theta_p}$ within each subQ. The following proposition explains why.

Proposition 6.2.1. Under any specific value $\boldsymbol{\theta_c^j}$, only subQ-level Pareto optimal solutions $(\boldsymbol{\theta_c^j}, \boldsymbol{\theta_p^*})$ contribute to the query-level Pareto optimal solutions.

The above result allows us to restrict our search of $\boldsymbol{\theta_p}$ to only the local Pareto optimal ones. However, given the large, diverse set of $\boldsymbol{\theta_c}$ candidates, it is computationally expensive to solve the MOO problem for $\boldsymbol{\theta_p}$ for each $\boldsymbol{\theta_c}$ candidate. We next introduce a clustering-based approximation to reduce the computation complexity.

It is based on the hypothesis that, within the same subQ, similar $\boldsymbol{\theta_c}$ candidates exhibit similar optimal $\boldsymbol{\theta_p}$ values in the tuning process. By clustering similar $\boldsymbol{\theta_c}$ values into a small number of groups (based on their Euclidean distance), we then solve the MOO problem of $\boldsymbol{\theta_p}$ for a single $\boldsymbol{\theta_c}$ representative of each group. To expedite the repeated solving of $\boldsymbol{\theta_p}$ for different $\boldsymbol{\theta_c}$ representatives, we maintain a pool of samples of $\boldsymbol{\theta_p}$ and among them find the Pareto optimal values for each $\boldsymbol{\theta_c}$ representative. We then use the optimized $\boldsymbol{\theta_p}$ as the estimated optimal solution for similar $\boldsymbol{\theta_c}$ candidates within each group.

Algorithm 2: Effective Set Generation

Require: Q, $\phi_i, n, \forall i \in [1,k], \alpha, \beta, \gamma, C, P$.

1: $\boldsymbol{\Theta_c^{(0)}}, \boldsymbol{\Theta_p^{(0)}}$ = sampling(C, P)
2: rep_c_list, C_list, κ = cluster($\boldsymbol{\Theta_c^{(0)}}$, n)
3: $\boldsymbol{\Theta_p^*}$ = optimize_p_moo($\boldsymbol{\Theta_p^{(0)}}$, rep_c_list, ϕ, α, β, γ, Q)
4: $\boldsymbol{\Omega^{(0)}}$, $\boldsymbol{\Theta^{(0)}}$ = assign_opt_p(C_list, rep_c_list, $\boldsymbol{\Theta_p^*}$, ϕ, α, β, γ, Q)
5: $\boldsymbol{\Theta_c^{(new)}}$ = enrich_c($\boldsymbol{\Omega^{(0)}}$, $\boldsymbol{\Theta^{(0)}}$)
6: $C\_list^{(new)}$ = assign_cluster($\boldsymbol{\Theta_c^{(new)}}$, rep_c_list, κ)
7: $\boldsymbol{\Omega^{(new)}}$, $\boldsymbol{\Theta^{(new)}}$ = assign_opt_p($C\_list^{(new)}$, rep_c_list, $\boldsymbol{\Theta_p^*}$, ϕ, α, β, γ, Q)
8: $\boldsymbol{\Omega}$, $\boldsymbol{\Theta}$ = union($\boldsymbol{\Omega^{(0)}}$, $\boldsymbol{\Theta^{(0)}}$, $\boldsymbol{\Omega^{(new)}}$, $\boldsymbol{\Theta^{(new)}}$)
9: **return** $\boldsymbol{\Omega}$, $\boldsymbol{\Theta}$

Algorithm. Algorithm 2 describes the steps for obtaining an effective solution set of $(\boldsymbol{\theta_c}, \boldsymbol{\theta_p})$ for each subQ. Line 1 initiates by generating $C \times P$ samples, where C and P are the numbers of distinct values of $\boldsymbol{\theta_c}$ and $\boldsymbol{\theta_p}$, respectively. The $\boldsymbol{\theta_c}$ candidates are then grouped using a clustering approach (Line 2), where rep_c_list constitutes the list of $\boldsymbol{\theta_c}$ representatives for the n groups, C_list includes the members within all n groups, and κ represents the clustering model. In Line 3, $\boldsymbol{\theta_p}$ optimization is performed for each representative $\boldsymbol{\theta_c}$ candidate (using the samples from Line 1). Subsequently, the optimal $\boldsymbol{\theta_p}$ of the representative $\boldsymbol{\theta_c}$ is assigned to all members within the same group and is fed to the predictive models to get objective values (Line 4). After that, the initial effective set is obtained, where $\Omega^{(0)}$ represents the subQ-level objective values under different $\boldsymbol{\theta_c}$, and $\Theta^{(0)}$ represents the corresponding configurations. Line 5 further enriches $\boldsymbol{\theta_c}$ using the crossover method or random sampling, which expands the initial effective set to generate new $\boldsymbol{\theta_c}$ candidates. Afterwards, the cluster model κ assigns the new $\boldsymbol{\theta_c}$ candidates with their group labels

(Line 6). The previous optimal $\boldsymbol{\theta_p}$ values are then assigned to the new members within the same group, resulting in their corresponding subQ-level values as the enriched set (Line 7). Finally, the initial set and the enriched set are combined as the final effective set of subQ tuning (Line 8).

Sampling methods. We next detail the sampling methods used in Line 1 of the algorithm. (1) We include basic sampling methods, including *random sampling* and *Latin-hypercube sampling* (LHS) [72] as a grid-search method. (2) To reduce dimensionality, we introduce feature importance score (FIS) based parameter filtering: we sort the parameters by the FIS value from the trained model and leverage the long-tail distribution to drop the parameters at the tail. Precisely, many parameters at the tail have a low cumulative FIS and we apply a threshold (e.g., 5% model loss) to remove them from sampling. (3) We further propose an adaptive grid search method with FIS-based parameter filtering. Given the sample budget (C or P), it goes down the FIS ranking list and progressively covers one more parameter, including its min, median and max values. If it reaches the budget before adding all parameters, it ignores those uncovered ones. Otherwise, it loops over the list again to add more sampled values of each parameter. (4) We conduct hyperparameter tuning to derive low, medium, and high values for C and P. We also employ a simple runtime adaptive scheme to adjust the sampling budget based on the predicted latency under the default configuration. See our tech report [65] for more details.

6.2.3 DAG Aggregation

DAG aggregation aims to recover query-level Pareto optimal solutions from subQ-level solutions. This task is a combinatorial MOO problem, as each subQ must select a solution from its non-dominated solution set while satisfying the $\boldsymbol{\theta_c}$ constraint, i.e., identical $\boldsymbol{\theta_c}$ configuration among all subQs. The complexity of this combinatorial problem can be exponential in the number of subQs. Our proposed approach below addresses this challenge by providing optimality guarantees and reducing the computation complexity.

Simplified DAG. A crucial observation that has enabled our efficient methods is that in our problem setting, the optimization problem over a DAG structure can be

simplified to an optimization problem over a list structure. This is due to our choice of analytical latency and cost metrics, where the query-level objective can be computed as the sum of subQ-level objectives, which applies to the analytical latency, IO cost, CPU cost, etc., as explained in the previous section. The functionality of a DAG can be simulated with a list structure for computing query-level objectives.

HMOOC1: Divide-and-Conquer. This method was contributed by our colleague. Under a fixed $\boldsymbol{\theta_c}$, i.e., satisfying the constraint inherently, we propose a divide-and-conquer method to compute the Pareto set of the simplified DAG, which is reduced to a list of subQs. The idea is to (repeatedly) partition the list into two halves, solve their respective subproblems, and merge their solutions to global Pareto optimal ones. The merge operation enumerates all the combinations of solutions of the two subproblems, sums up their objective values and retains only the Pareto optimal ones.

This DAG aggregation method is described in Algorithm 3. The Ω, Θ are the effective set of all subQs, where Ω represents subQ-level objective values, and Θ represents the corresponding configurations, including $\{\boldsymbol{\theta}_c, \{\boldsymbol{\theta_p}\}, \{\boldsymbol{\theta_s}\}\}$. If there is only one subQ, it returns the Ω, Θ (lines 1-2). Otherwise, it follows a divide-and-conquer framework (lines 4-8).

The main idea is a merging operation, which is described in Algorithm 4. The input includes the subQ-level objective values (e.g. $\mathcal{F}^h$ is a Pareto frontier) and its configurations ($\mathcal{C}^r$) for the two nodes to be merged, where h and r denote they are two different nodes. It merges two nodes into a pseudo node by enumerating all the combinations of solutions in the two nodes (lines 2-3), summing up their objective values (lines 4-5) and taking its Pareto frontier as the solutions of this pseudo node (line 8).

HMOOC2: WS-based Approximation. We propose a second technique to approximate the MOO solution over a list structure. For each fixed $\boldsymbol{\theta_c}$, we apply the weighted sum (WS) method to generate evenly spaced weight vectors. Then for each weight vector, we obtain the (single) optimal solution for each subQ and sum the solutions of subQ's to get the query-level optimal solution. It can be proved that this WS method over a list of subQs guarantees to return a subset of query-level

Algorithm 3: General_Divide_and_conquer

Require: subQ-level values Ω, subQ-level configurations Θ.

1: **if** $|\Omega| == 1$ **then**
2: **return** Ω, Θ
3: **else**
4: Ω^h, Θ^h = first_half(Ω, Θ)
5: Ω^r, Θ^r = second_half(Ω, Θ)
6: $\mathcal{F}^h, \mathcal{C}^h$ = General_Divide_and_conquer(Ω^h, Θ^h)
7: $\mathcal{F}^r, \mathcal{C}^r$ = General_Divide_and_conquer(Ω^r, Θ^r)
8: **return** merge($\mathcal{F}^h, \mathcal{C}^h, \mathcal{F}^r, \mathcal{C}^r$)
9: **end if**

Algorithm 4: merge

Require: $\mathcal{F}^h, \mathcal{C}^h, \mathcal{F}^r, \mathcal{C}^r$.

1: $\mathcal{F}, \mathcal{C} = \emptyset, \emptyset$
2: **for** $(F_1, F_2), (c_1, c_2) \in (\mathcal{F}^h, \mathcal{C}^h)$ **do**
3: **for** $(F_1', F_2'), (c_1', c_2') \in (\mathcal{F}^r, \mathcal{C}^r)$ **do**
4: $\mathcal{F} = \mathcal{F} \cup \{(F_1 + F_1', F_2 + F_2')\}$
5: $\mathcal{C} = \mathcal{C} \cup \{(c_1, c_1'), (c_2, c_2')\}$
6: **end for**
7: **end for**
8: **return** $\mathcal{F}^*, \mathcal{C}^*$ = filter_dominated($\mathcal{F}, \mathcal{C}$)

Pareto solutions.

Algorithm 5 describes the full procedures. The input includes a *subQ_list*, which includes both subQ-level objective values and the corresponding configurations of all subQs. *ws_paris* are the weight pairs, e.g. $[[0.1, 0.9], [0.2, 0.8], \dots]$ for latency and cost. Line 1 initializes the query-level objective values and configurations. Lines 3-9 address the Weighted Sum (WS) method to generate the query-level optimal solution for each weight pair. Specifically, Lines 5-8 apply the WS method to obtain the optimal solution choice for each subQ, and sum all subQ-level values to get the query-level values. Upon iterating through all weights, a Pareto solution set is derived after the necessary filtering (Line 12).

HMOOC3: Boundary-based Approximation. Given that DAG aggregation under each $\boldsymbol{\theta_c}$ candidate operates independently, it is inefficient to do so repeatedly when we have a large number of $\boldsymbol{\theta_c}$ candidates. Our next approximate technique stems from the idea that the objective space of DAG aggregation under each $\boldsymbol{\theta_c}$ can be approximated by k *extreme points*, where k is the number of objectives. In this

Algorithm 5: Compressing_list_nodes

Require: subQ_list, ws_pairs.

```
PO = [ ], conf = [ ]
for [w_l, w_c] in ws_pairs do
    po_n = [ ], conf_n = {}
    for subQ_po, subQ_conf in subQ_list do
        subQ_po_norm = normalize_per_subQ(subQ_po)
        opt_ind = minimize_ws([w_l, w_c], subQ_po_norm)
        po_n.append(subQ_po[opt_ind])
        conf_n.append(subQ_conf[opt_ind])
    end for
    PO.append(sum(po_n)), conf.append(conf_n)
end for
return filter_dominated(PO, conf)
```

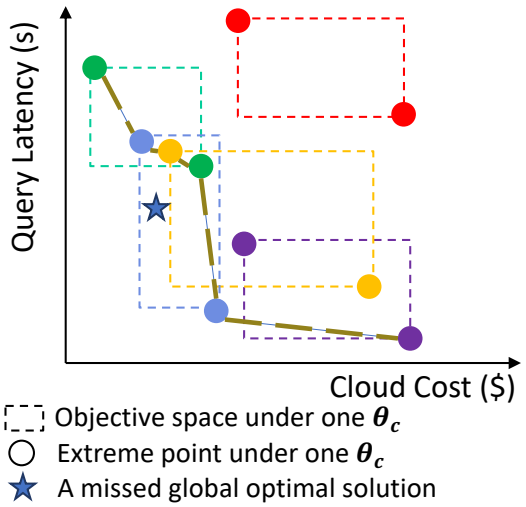


Figure 6.7: Approximated DAG optimization

context, the *extreme point* under a fixed $\boldsymbol{\theta_c}$ is the Pareto optimal point with the best query-level value for any objective. Then, the approximate query-level Pareto set is determined by the non-dominated extreme points among all $\boldsymbol{\theta_c}$ points.

The rationale behind this approximation lies in the observation that solutions from different $\boldsymbol{\theta_c}$ candidates correspond to distinct regions on the query-level Pareto front. This arises from the fact that each $\boldsymbol{\theta_c}$ candidate determines the total resources allocated to the query, and a diverse set of $\boldsymbol{\theta_c}$ candidates ensures good coverage across these resources. Varying total resources, in turn, lead to different objectives of query performance, hence resulting in good coverage of the Pareto front of cost-performance tradeoffs.

Therefore, we consider the degenerated *extreme points* to symbolize the bound-

aries of different (resource) regions within the query-level Pareto front. Figure 6.7 illustrates an example. Here, extreme points and the dashed rectangles of different colors represent the query-level objective space of Pareto optimal solutions under various $\boldsymbol{\theta_c}$ candidates. The brown dashed line represents the approximate query-level Pareto front derived by filtering the dominated solutions from the collection of extreme points. The star solution indicates a missed query-level Pareto solution, as it cannot be captured from the extreme points.

The algorithm works as follows. For each $\boldsymbol{\theta_c}$ candidate, for each objective, we select the subQ-level solution with the best value for that objective for each subQ, and then sum up the objective values of such solutions from all subQs to form one *extreme point*. Repeating this procedure will lead to a maximum of kn query-level solutions, where k is the number of objectives and n is the number of $\boldsymbol{\theta_c}$ candidates. An additional filtering step will retain the non-dominated solutions from the kn candidates, using an existing method of complexity $O(kn \log(kn))$ [50].

Our formal results include the following, and more details of theoretical analysis in compile-time optimization are discussed in Section 6.3.

Proposition 6.2.2. Under a fixed $\boldsymbol{\theta_c}$ candidate, the query-level objective space of Pareto optimal solutions is bounded by its extreme points in a 2D objective space.

Proposition 6.2.3. Given subQ-level solutions, our boundary approximation method guarantees to include at least k query-level Pareto optimal solutions for a MOO problem with k objectives.

6.2.3.1 Multiple Query Plan Search

Our compile-time MOO algorithm so far has considered only one physical query plan (with the corresponding subQs) based on the default configuration. Since at runtime, AQE may generate a very different physical plan, we further enhance our compile-time optimization by considering multiple physical plans. Initially, we sample plan-related parameters (s_3 and s_4) in $\boldsymbol{\theta}_p$ to collect different physical query plans. We then rank these plans based on their predicted query latency under the default configuration. Subsequently, we trigger compile-time optimization (the HMOOC algorithm) for each of the top-k fastest query plans, run them in parallel,

and merge their Pareto solutions into the same objective space to obtain the final Pareto set.

6.3 Theoretical Analysis

In this section, we include the algorithms, proofs and complexity analysis of our optimization techniques.

6.3.1 Subquery (subQ) Tuning

Proposition 6.2.1 Under any specific value $\boldsymbol{\theta_c^j}$, only subQ-level Pareto optimal solutions $(\boldsymbol{\theta_c^j}, \boldsymbol{\theta_{pi}^*})$ for the i-th subQ contribute to the query-level Pareto optimal solutions $(\boldsymbol{\theta_c^j}, \{\boldsymbol{\theta_p^*}\})$.

Proof. Let F_q^j be a query-level Pareto optimal solution for $\boldsymbol{\theta_c^j}$. It can be expressed as $F_q^j = \sum_{i=1}^m F_{si}^j$. Assume that there exists at least one i, e.g., i_1, such that $F_{si_1}^j$ is not optimal for the i_1-th subQ. Let $F_q^{j\prime} = F_{si_1}^{j\prime} + \sum_{i=1, i\neq i_1}^m F_{si}^j$ where $F_{si_1}^{j\prime}$ is Pareto optimal for the i_1-th subQ.

We have

$$F_q^j - F_q^{j\prime} = F_{si_1}^j - F_{si_1}^{j\prime} \tag{6.3}$$

Since $F_{si_1}^{j\prime}$ is optimal for the i_1-th subQ, we have $F_{si_1}^j > F_{si_1}^{j\prime}$. This means that $F_{si_1}^j$ is dominated by $F_{si_1}^{j\prime}$, which contradicts our hypothesis. Therefore, a Pareto optimal solution for the query-level can only contain subQ-level Pareto optimal solutions under a fixed $\boldsymbol{\theta_c^j}$.

□

Complexity analysis of model inference related to the sampling rate. The sampling rate includes the number of $\boldsymbol{\theta_c}$ and $\boldsymbol{\theta_p}$ samples, denoted as N_c and N_p respectively. The model is called three times in line 3, 4 and 7 respectively.

Line 3 calls the model to provide the predictions of C cluster-representative $\boldsymbol{\theta_c}$ and N_p $\boldsymbol{\theta_p}$ samples. For m subQs, there is $m \times C \times N_p$ predictions. It filters the dominated solutions of each $\boldsymbol{\theta_c}$ under each subQ to ensure that $\boldsymbol{\theta_c}$ is identical among all subQs while maintaining the optimality of $\boldsymbol{\theta_p}$, as illustrated in Proposition 6.2.1.

Line 4 assigns the optimal $\boldsymbol{\theta_p}$ solutions to N_c initialized $\boldsymbol{\theta_c}$ samples for all subQs and provides the corresponding predictions. Since the number of optimal solutions of a $\boldsymbol{\theta_c}$ sample for different subQs could be different, for simplicity, we use p_{avg} to denote the average number of optimal solutions among all $\boldsymbol{\theta_c}$ samples and all subQs. The total number of predictions is $N_c \times p_{avg} \times m$.

Line 7 functions the same as line 4, where the only difference is to assign the optimal $\boldsymbol{\theta_p}$ solutions (returned from line 3) to the newly generated $\boldsymbol{\theta_c}$ samples for all subQs. Assume N_c^{new} new $\boldsymbol{\theta_c}$ samples are generated, the total number of predictions is $N_c^{new} \times p_{avg} \times m$.

6.3.2 DAG Aggregation

We provide further details of three methods for DAG aggregation.

HMOOC1: Divide-and-Conquer. From the perspective of optimality, given the effective set of subQ-level solutions, Algorithm 3 is proven to return a complete set of query-level Pareto optimal solutions, as it enumerates over all subQ-level solutions.

The complexity of *merge* function is $O(M*N)+O((M*N)log(M*N))$ if there are M and N solutions in two nodes, where the enumeration takes $O(M*N)$ and filtering dominated solutions takes $O((M*N)log(M*N))$. However, after multiple merging operations, both M and N could become large, leading to a potentially high total complexity.

The core operation in HMOOC1 is the *merge*, which enumerates over all subQ-level solutions. The following are the theoretical proof. For the sake of simplicity, we consider the case with two nodes.

Proposition 6.3.1. Algorithm 3 always output the full Pareto front of the simplified DAG.

Proof. Let D and G be two nodes (e.g., subQs or aggregated subQs). Let $\oplus$ be the Minkowski sum, i.e., $D \oplus G = \{F_D + F_G, F_D \in D, F_G \in G\}$. Let Pf denote the Pareto Front of a node.

$$Pf(Pf(D) \oplus Pf(G)) = Pf(D \times G) \tag{6.4}$$

Let $E_D : \mathcal{C}_D \to \mathbb{R}$ be the evaluation function of node D, where $\mathcal{C}_D$ is the set of configurations for node D. We define E_G in a similar manner. We also define $E : (c_D, c_G) \mapsto E_D(c_D) + E_G(c_G)$, where c_D and c_G are one configuration in $\mathcal{C}_D$ and $\mathcal{C}_G$ respectively.

Let $p \in Pf(Pf(D) \oplus Pf(G))$. Then p can be addressed as a sum of two terms: one from $Pf(D)$ and the other from $Pf(G)$, i.e.,

$$p = p_D + p_G, \qquad p_D \in Pf(D), p_G \in Pf(G) \tag{6.5}$$

Let $c_D \in Pf^{-1}(p_D)$ and $c_G \in Pf^{-1}(p_G)$, i.e., c_D is chosen such as $E_D(c_D) = p_D$, and likewise for c_G. Then we have $p = E(c_D, c_G)$, so p belongs to the objective space of $D \times G$. Suppose p doesn't belong to $Pf(D \times G)$. This means that there exists p' in $Pf(D \times G)$ that dominates p. p' can be expressed as $p' = p'_D + p'_G$ with $p'_D \in Pf(D)$ and $p'_G \in Pf(G)$. p' dominates p can be expressed as $p'_D < p_D$ and $p'_G \leq p_G$ or $p'_D \leq p_D$ and $p'_G < p_G$ in our optimization problem with minimization, which contradicts the definition of p as belonging to the Pareto Front. Therefore p must belong to $Pf(D \times G)$ and thus $Pf(Pf(D) \oplus Pf(G)) \subseteq Pf(D \times G)$.

Let us now suppose that $p \in Pf(D \times G)$. Then there exists c in $D \times G$, i.e., $c = (c_D, c_G)$ with $c_D \in D, c_G \in G$, such that $p = E(c)$. By setting $p_D = E_D(c_D)$ and $p_G = E_G(c_G)$, we have $p = p_D + p_G$. By definition, p_D belongs to $Pf(D)$ and p_G belongs to $Pf(G)$. Hence, $p \in Pf(D) \oplus Pf(G)$.

Suppose that p doesn't belong to $Pf(Pf(D) \oplus Pf(G))$ and that there exists p' in $Pf(Pf(D) \oplus Pf(G))$ that dominates p. We showed above that p' must belong to $Pf(D \times G)$. Thus both p and p' belong to $Pf(D \times G)$ and p' dominates p, which is impossible. Therefore p' must belong to $Pf(Pf(D) \oplus Pf(G))$ and $Pf(Pf(D) \oplus Pf(G)) \subseteq Pf(D \times G)$.

By combining the two inclusions, we obtain that $Pf(Pf(D) \oplus Pf(G)) = Pf(D \times G)$, where the left side is the Pareto optimal solution from the Algorithm 3 and the right side is the Pareto optimal solution over the whole configuration space of two nodes. Thus, Algorithm 3 returns a full set of Pareto solutions.

□

HMOOC2: WS-based Approximation.

Lemma 6.3.1. *For a DAG aggregation problem with k objectives that use the* `sum` *operator only, Algorithm 5 guarantees to find a non-empty subset of query-level Pareto optimal points under a specified $\boldsymbol{\theta_c}$ candidate.*

In proving Lemma 6.3.1, we observe that Algorithm 5 is essentially a Weighted Sum procedure over Functions (WSF). Indeed we will prove the following two Propositions: 1) each solution returned by WSF is Pareto optimal; 2) the solution returned by the Algorithm 5 is equivalent to the solution returned by WSF. Then it follows that the solution returned by Algorithm 5 is Pareto optimal.

To introduce WSF, we first introduce the indicator variable x_{ij}, $i \in [1, ..., m], j \in [1, ..., p_i]$, to indicate that the j-th solution in i-th subQ is selected to contribute to the query-level solution. $\sum_{j=1}^{p_i} x_{ij} = 1$ means that only one solution is selected for each subQ. Then $x = [x_{1j_1}, ...x_{mj_m}]$ represents the 0/1 selection for all m subQs to construct a query-level solution. Similarly, $f = [f^1_{1j_1}, \ldots, f^k_{mj_m}]$ represents the value of the objectives associated with x.

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

$$\begin{aligned} F_v = H_v(x; f) = \sum_{i=1}^{m} \sum_{j=1}^{p_i} x_{ij} \times f^v_{ij}, \\ where \sum_{j=1}^{p_i} x_{ij} = 1, i \in [1, ..., m], v \in [1, ...k] \end{aligned} \tag{6.6}$$

For simplicity, we refer to $H_v(x; f)$ as $H_v(x)$ when there is no confusion. Now we introduce the Weighted Sum over Functions (WSF) as:

$$\operatorname*{argmin}_x (\sum_{v=1}^{k} w_v \times H_v(x)) \tag{6.7}$$

$$\text{s.t.} \quad \sum_{v=1}^{k} w_v = 1, \quad w_v \geq 0 \text{ for } v = 1, \ldots, k \tag{6.8}$$

where w_v is the weight value for objective v. Next, we prove for Lemma 6.3.1. As

stated before, It is done in two steps.

Proposition 6.3.2. The solution constructed using x returned by WSF is Pareto optimal.

Proof.

Assume that x^* (corresponding to $[F_1^*, ..., F_k^*]$) is the solution of WSF. Suppose that another solution $[F_1^{'}, ..., F_k^{'}]$ (corresponding to x') dominates $[F_1^*, ..., F_k^*]$. This means that $\sum_{v=1}^{k} w_v \times H_v(x')$ is smaller than that of x^*.

This contradicts that x^* is the solution of WSF. So there is no $[F_1^{'}, ..., F_k^{'}]$ dominating $[F_1^*, ..., F_k^*]$. Thus, $[F_1^*, ..., F_k^*]$ is Pareto optimal. □

Proposition 6.3.3. The optimal solution returned by the Algorithm 5 is equivalent to the solution constructed using x returned by WSF.

Proof.

Suppose x' is returned by WSF. The corresponding query-level solution is $[F_1^{'}, ..., F_k^{'}]$

$$\begin{aligned} x' &= \operatorname{argmin}_x(\sum_{v=1}^{k} w_v \times H_v(x)) \\ &= \operatorname{argmin}(\sum_{v=1}^{k} w_v \times (\sum_{i=1}^{m}\sum_{j=1}^{p_i} x_{ij} \times f_{ij}^v)) \\ &= \operatorname{argmin}(\sum_{i=1}^{m}(\sum_{v=1}^{k}\sum_{j=1}^{p_i}(w_v \times f_{ij}^v) \times x_{ij})) \end{aligned} \tag{6.9}$$

For the solution $[F_1^{''}, ..., F_k^{''}]$ returned by Algorithm 5, x'' represents the corresponding selection. It is achieved by minimizing the following formula:

$$\begin{aligned} &\sum_{i=1}^{m} \min_{j\in[1,p_i]}(WS_{ij}) \\ &= \sum_{i=1}^{m} \min_{j\in[1,p_i]}(\sum_{v=1}^{k} w_v \times f_{ij}^v) \\ &= \sum_{i=1}^{m} \min_{j\in[1,p_i]}(\sum_{v=1}^{k}\sum_{j=1}^{p_i}(w_v \times f_{ij}^v) \times x_{ij}) \end{aligned} \tag{6.10}$$

where $WS_{ij} = \sum_{v=1}^{k} w_v \times f_{ij}^v$. Given a fixed i, x_{ij} can only be positive (with value 1) for one value of j.

So, x'' must solve:

$$\begin{aligned} x'' &= (\sum_{i=1}^{m} \underset{x_{ij}}{\operatorname{argmin}} (\sum_{v=1}^{k} \sum_{j=1}^{p_i} (w_v \times f_{ij}^v) \times x_{ij})) \\ &= \underset{x}{\operatorname{argmin}} (\sum_{i=1}^{m} (\sum_{v=1}^{k} \sum_{j=1}^{p_i} (w_v \times f_{ij}^v) \times x_{ij})) \end{aligned} \tag{6.11}$$

Here, optimizing for each subQ is independent of the optimization of the other subQs, so we can invert the sum over i and the arg min. Thus, $x' = x''$. Therefore, WSF and Algorithm 5 are equivalent.

□

With these two propositions, we finish the proof of Lemma 6.3.1.

Algorithm 5 varies w weight vectors to generate multiple query-level solutions. And under each weight vector, it takes $O(m \cdot p_{max})$ to select the optimal solution for each LQP-subtree based on WS, where p_{max} is the maximum number of solutions among m subQs. Thus, the overall time complexity of one $\boldsymbol{\theta_c}$ candidate is $O(w \cdot (m \cdot p_{max}))$.

HMOOC3: Boundary-based Approximation.

Proposition 6.2.2 Under a fixed $\boldsymbol{\theta_c}$ candidate, the query-level objective space of Pareto optimal solutions is bounded by its extreme points in a 2D objective space.

Proof.

Assume that $F_q^1 = [F_q^{1*}, F_q^{2-}]$ and $F_q^2 = [F_q^{1-}, F_q^{2*}]$ are two *extreme points* under a fixed $\boldsymbol{\theta_c}$, recalling that the *extreme point* under a fixed $\boldsymbol{\theta_c}$ is the Pareto optimal point with the best query-level value for any objective. Here the superscript $\{1*\}$ means it achieves the best in objective 1 and $\{2*\}$ means it achieves the best in objective 2. The two extreme points form an objective space as a rectangle.

Suppose that an existing query-level Pareto optimal solution $F_q' = [F_q^{1'}, F_q^{2'}]$ is outside this rectangle, which includes 2 scenarios. In scenario 1, it has $F_q^{1'} < F_q^{1*}$ or $F_q^{2'} < F_q^{2*}$, which is impossible as extreme points already achieves the minimum

values of two objectives. In scenario 2, it has $F_q^{1'} > F_q^{1-}$ or $F_q^{2'} > F_q^{2-}$, which is impossible as $F_q^{'}$ is dominated by any points inside the rectangle.

So there is no Pareto optimal solution $F_q^{'}$ existing outside the rectangle and it concludes the proof. □

Proposition 6.2.3 Given subQ-level solutions, our boundary approximation method guarantees to include at least k query-level Pareto optimal solutions for a MOO problem with k objectives.

Proof.

Assume that $F_q^1, ...F_q^k$ are k extreme points, which are Pareto optimal and achieve the best (e.g. the lowest in the minimization problem) query-level values of objectives $1, ..., k$ among all $\boldsymbol{\theta_c}$ configurations. Suppose that an existing Pareto optimal solution $F_q^{'}$, distinct from the extreme points, dominates any point in $F_q^1, ...F_q^k$. $F_q^{'}$ must achieve a better value than $F_q^{1*}, ...F_q^{k*}$ in any objectives $1, ..., k$, where the superscript $\{1*\}$ means it achieves the best in objective 1 and $\{k*\}$ means it achieves the best in objective k. which is impossible as the extreme points already achieves the best.

So these k extreme points cannot be dominated by any other solutions. Thus, they are Pareto optimal and this concludes the proof. □

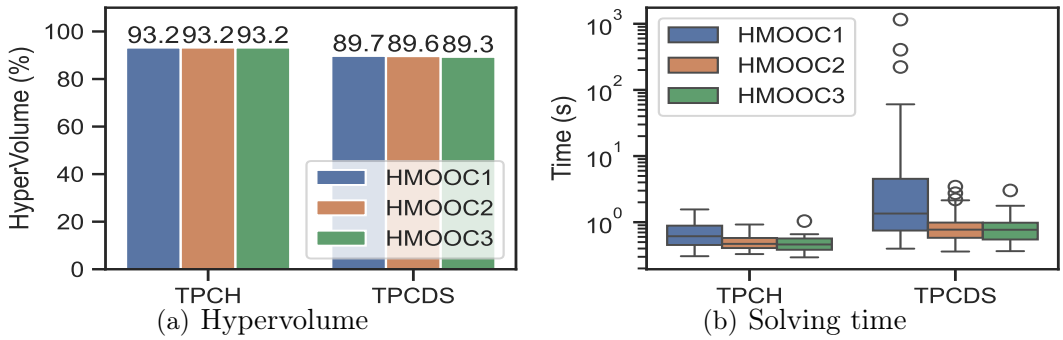
6.4 Experimental Evaluation

In this section, we evaluate our fine-grained compile-time optimization techniques.

6.4.1 Setup

Spark setup. We perform SQL queries at two 6-node Spark 3.5.0 clusters with runtime optimization plugins. Our optimization focuses on 19 parameters, including 8 for $\boldsymbol{\theta}_c$, 9 for $\boldsymbol{\theta}_p$, and 2 for $\boldsymbol{\theta}_s$, selected based on feature selection profiling [40] and best practices from the Spark documentation. More details are in tech report [65].

Workloads. We generate datasets from the TPC-H and TPC-DS benchmarks with a scale factor of 100. We use the default 22 TPC-H and 102 TPC-DS queries for the



(a) Hypervolume (b) Solving time

Figure 6.8: Comparison of performance of all DAG aggregation methods

optimization analyses and end-to-end evaluation. To collect training data, we further treat these queries as templates to generate 50k distinct parameterized queries for TPC-H and TPC-DS, respectively. We run each query under one configuration sampled via Latin Hypercube Sampling [72].

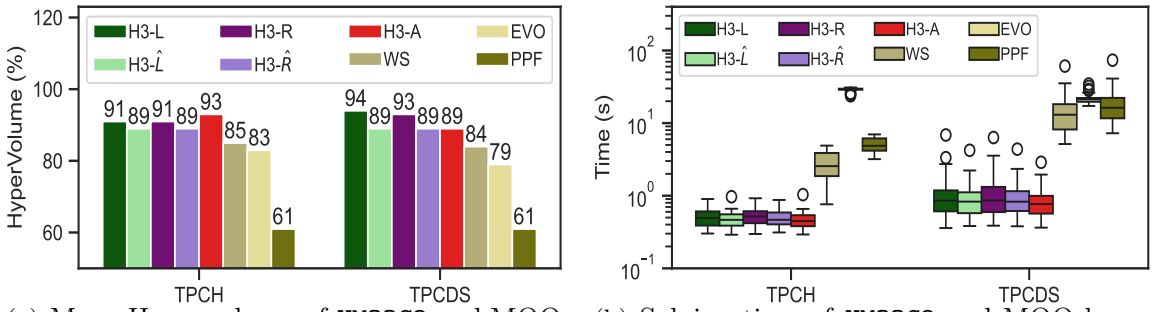
Implementation. We implement our optimizer using plugins into Spark. (1) The trace collector is implemented as customized Spark listeners to track runtime plan structures and statistics, costing an average of 0.02s overhead per query. (2) The compile-time optimizer operates as a standalone module, costing an average of 0.4s. (3) The model server maintains up-to-date models on the same server as the optimizer. The TPC-H and TPC-DS traces were split into 8:1:1 for training, validation, and testing. It took 6-12 hours to train one model and 2 weeks for hyperparameter tuning on a GPU node with 4 NVIDIA A100 cards.

6.4.2 Compile-time MOO Methods

We next evaluate our compile-time MOO methods against existing MOO methods. The objectives include query latency and cloud cost as a weighted sum of cpu hours, memory hours, shuffle sizes.

Expt 1: *DAG Aggregation methods.* We first compare the three DAG aggregation methods (§6.2.3) in the HMOOC framework in terms of accuracy and efficiency. Hypervolume (HV) is a standard measure of the dominated space of a Pareto set in the objective space. All three methods provide similar HV but varies in solving time. Figures 6.8 shows that Boundary-based Approximation (`HMOOC3`) is the most efficient for both benchmarks, achieving the mean solving time of 0.5-0.8s. Therefore, we use

HMOOC3 in the remaining experiments.



(a) Mean Hypervolume of HMOOC3 and MOO baselines

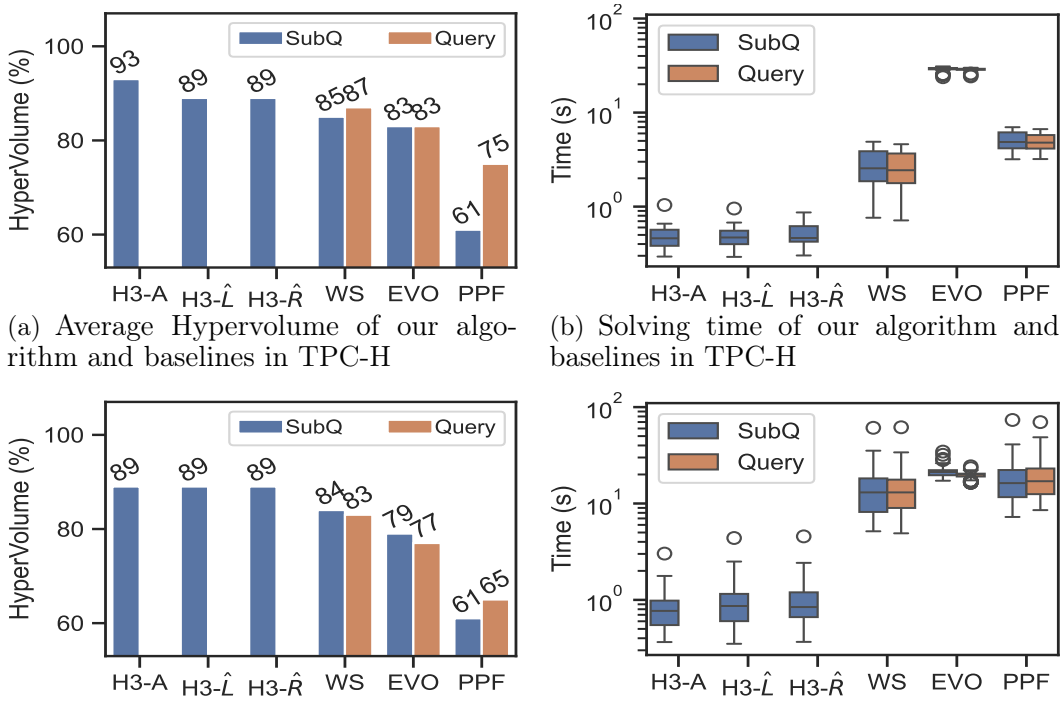
(b) Solving time of HMOOC3 and MOO baselines

Figure 6.9: Accuracy and efficiency of our compile-time MOO algorithms, compared to existing MOO methods

Expt 2: *Sampling Methods.* We next compare different sampling methods used with HMOOC3, where H3-R and H3-L denote Random sampling and Latin Hypercube Sampling (LHS), and H3-$\hat{R}$ and H3-$\hat{L}$ denote their variants with feature importance score (FIS) based parameter filtering. Further, H3-A denotes our Adaptive grid-search with FIS-based parameter filtering. The sampling rate is set uniformly ($C = 54$) $\times$ ($P = 243$) for all methods. Figures 6.9 report on HV and solving time, in the first 5 bars in each group. In terms of HV, H3-A is the best for TPC-H, and H3-L and H3-R slightly outperform the other three with parameter filtering for TPC-DS. Considering solving time, we see that H3-A achieves the lowest solving time, finishing all TPC-H queries in 1s and 100/102 TPC-DS queries in 2s. The methods without parameter filtering can have the solving time exceeding 6 seconds for some TPC-DS queries. Overall, H3-A achieves a good balance between HV and solving time.

Expt 3: *Comparison with SOTA MOO methods.* We next compare with SOTA MOO methods, WS [69] (with tuned hyperparameters of 10k samples and 11 pairs of weights), Evo [23] (with a population size of 100 and 500 function evaluations), and PF [86], for fine-grained tuning of parameters based on Definition 6.1.1. Figures 6.9 report their HV and solving time, in the last three bars of each group. HMOOC3 outperforms other MOO methods with 4.7%-54.1% improvement in HV and 81%-98.3% reduction in solving time. These results stem from HMOOC's hierarchical framework, which addresses a smaller search space with only one set of $\boldsymbol{\theta_c}$ and $\boldsymbol{\theta_p}$ at a time, and uses efficient DAG aggregation to recover query-level values from

subQ-level ones. In contrast, other methods solve the optimization problem using the global parameter space, including m sets of $\boldsymbol{\theta_p}$, where m is the number of subQs in a query.



(a) Average Hypervolume of our algorithm and baselines in TPC-H

(b) Solving time of our algorithm and baselines in TPC-H

(c) Average Hypervolume of our algorithm and baselines in TPC-DS

(d) Solving time of our algorithm and baselines in TPC-DS

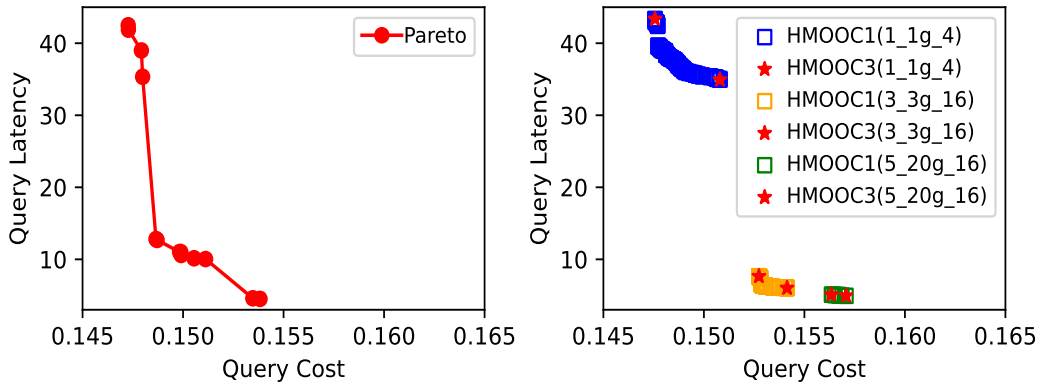
Figure 6.10: Analytical performance of our algorithm, compared to the state-of-the-art (SOTA) methods with query-level tuning

Expt 4: *Comparison with Query-level Tuning.* Figures 6.10 show results of query-level tuning, where TPC-H queries take over an average of at least 2.7s and much lower HV (at most 87%) than `HMOOC3` (93%). All of TPCDS queries with query-level tuning lose to `HMOOC3` in HV (at most 83% v.s. 89%) and in solving time (the average exceeding 14s v.s. 0.8s).

Additional end-to-end experimental results provided by our colleague can be found in our technical report [65], which extends our best compile-time optimization method, `HMOOC3`, to include runtime optimization.

6.4.3 Breakdown Analysis

This section provides a breakdown analysis of DAG aggregation methods, the necessity of finer control of parameters, and the enrichment of new $\boldsymbol{\theta_c}$.



(a) Pareto frontier of TPCH Q2

(b) Pareto frontier of TPCH Q2 with different total resources

Figure 6.11: Objective space under differen resources

Analysis on DAG aggregation methods. `HMOOC1` and `HMOOC3` achieve comparable hypervolume because they have the similar coverage of the Pareto frontiers in the objective space. To verify this, $\boldsymbol{\theta_c}$ candidates are implemented with three different total resource settings, as shown in Figure 6.11. For example, the setting 1_1g_4 indicates the values of parameters k_1, k_2, and k_3, representing the number of cores per executor (e.g., 1), the memory size per executor (e.g., 1g), and the number of executors (e.g., 4), respectively. Figure 6.11 demonstrates that solutions from diverse total resources cover various regions of the Pareto frontier, and `HMOOC3` achieves the boundary points of `HMOOC1`. It is noteworthy that all DAG aggregation methods have the same effective set. Thus, `HMOOC3` is able to achieve similar coverage of the Pareto frontier to that from `HMOOC1` when $\boldsymbol{\theta_c}$ candidates within the effective set cover diverse total resources.

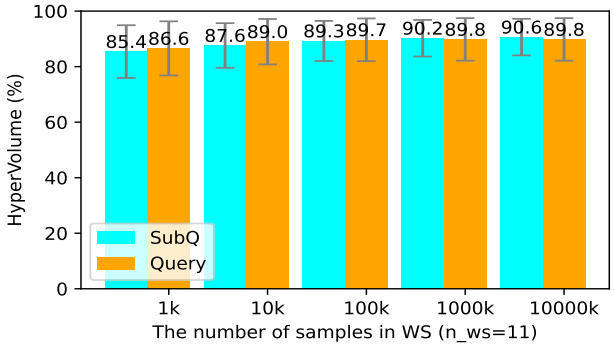


Figure 6.12: Comparison of query-control and finer-control with smaller searching space

Analysis on necessity of finer-control. It's noteworthy that query-control can-

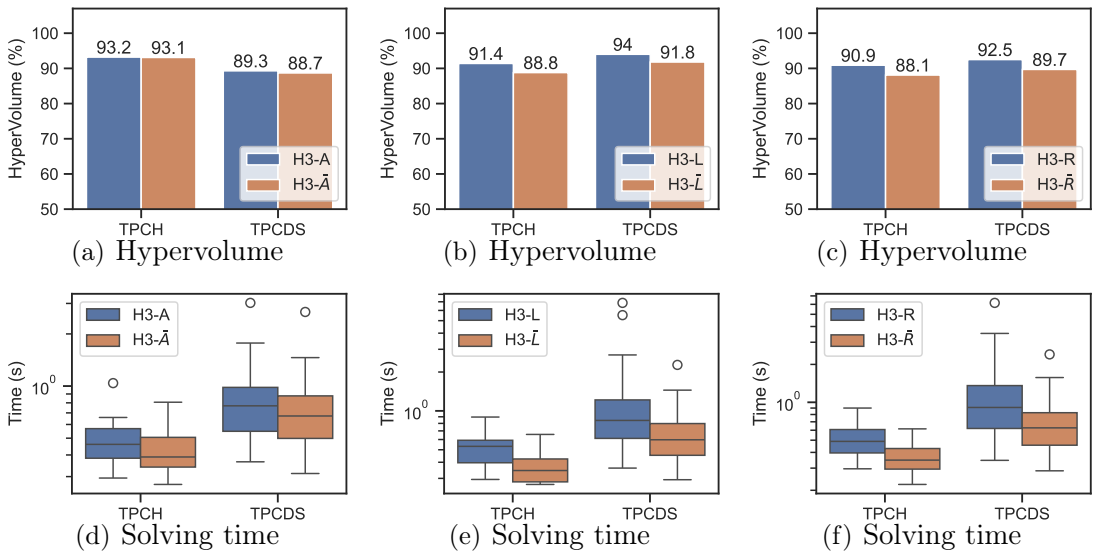
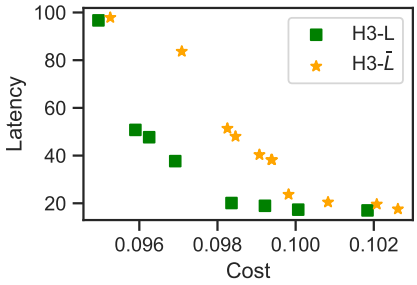


Figure 6.13: Comparison of Hypervolume and solving time with/without new $\boldsymbol{\theta_c}$ extension for three sampling methods

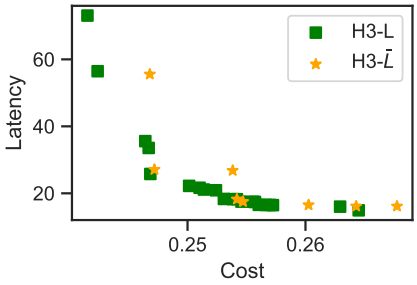
not achieve a higher upper bound than finer-control. To verify this, we implemented a smaller search space (each parameter having only 2 values) for WS to fully explore query-control, where WS performs the best among all baselines for both TPC-H and TPC-DS. Figure 6.12 displays the hypervolumes (HVs) of WS under different numbers of samples, with blue and orange bars representing the HVs of finer-control and query-control, respectively. It is observed that as the number of samples increases, the HV of query-control stops increasing at 1M samples (89.8%), while the HV of finer-control continues to improve (90.6%). This demonstrates that finer-control has the potential to achieve better solutions than query-control, illustrating the necessity of finer-control in our problem.

Analysis on new $\boldsymbol{\theta_c}$ extension. Figure 6.13 compares HV and solving time with and without new $\boldsymbol{\theta_c}$ extension for all sampling methods, where `H3-`$\bar{A}$, `H3-`$\bar{L}$ and `H3-`$\bar{R}$ denote results without new $\boldsymbol{\theta_c}$ extension from adaptive grid search, LHS and random sampling.

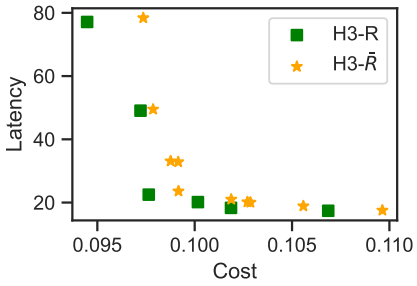
For adaptive grid search, it extends new $\boldsymbol{\theta_c}$ by random sampling. Figures 6.13(a) and 6.13(d) show the HV and solving time of all queries in TPC-H and TPC-DS. Without new $\boldsymbol{\theta_c}$ extension, HV reduces slightly compared to the HV with new $\boldsymbol{\theta_c}$ extension ($\leq 0.6\%$). This is because the adaptive grid search uses random sampling to extend new $\boldsymbol{\theta_c}$, which works to extend more global Pareto optimal $\boldsymbol{\theta_c}$ while with



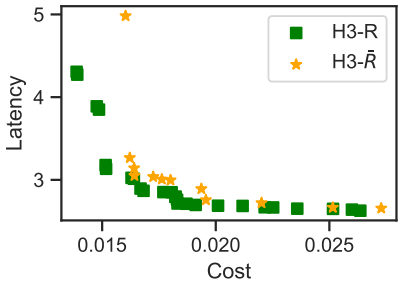
(a) TPCH-Q1 with LHS



(b) TPCDS-Q78 with LHS



(c) TPCH-Q1 with random sampling



(d) TPCDS-Q13 with random sampling

Figure 6.14: Comparison of Pareto frontiers with/without crossover in LHS and random-sampling

limit capability due to the high-dimensional space. Figures 6.13(b) and 6.13(e) compare the performance of LHS with and without the new $\boldsymbol{\theta_c}$ extension by crossover. These figures display the HV and solving time for all queries in TPC-H and TPC-DS. Without crossover, HV decreases by up to 2.6% compared to HV with crossover. The HV reduction is not substantial because boundary solutions (i.e., those with minimum latency or minimum cost) have a significant impact on HV by defining the boundaries of the dominated space on the Pareto front. Meanwhile, crossover contributes to generating better global optimal solutions within the middle region of the Pareto front, as illustrated in Figures 6.14. The solving time without crossover decreases as it reduces the procedures following the new $\boldsymbol{\theta_c}$ extension. Similar observations are seen in Figures 6.13(c) and 6.13(f) for extending new $\boldsymbol{\theta_c}$ in random sampling.

Figures 6.14 show the Pareto frontiers of example queries based on the maximum HV differences between results with and without crossover using LHS and random sampling. Green points represent solutions with the crossover operation, while orange points represent solutions without crossover. For both LHS and random sam-

pling as the initial sampling methods, solutions with crossover yield better results in the middle region of the Pareto front. This improvement is due to the crossover location set in the implementation. In our experiment, the crossover location is set to 3 based on hyperparameter tuning, which splits $\boldsymbol{\theta_c}$ into two sets: resource and non-resource parameters. The Cartesian product of these two sets does not generate new values for total resources. Consequently, the crossover explores better solutions in the middle region rather than better boundary solutions with significantly lower latency or cost.

6.5 Summary

This chapter presented an efficient MOO algorithm designed for fine-grained Spark parameter tuning with complex control, aiming to achieve Pareto optimality for the entire query. Our approach decomposes the optimization problem of a large parameter space into smaller subproblems, each constrained to use the same shared parameters. Evaluation results using the TPC-H and TPC-DS benchmarks demonstrate that `HMOOC` outperforms existing MOO methods, achieving improvements in hypervolume ranging from 4.7% to 54.1% and reductions in solving time ranging from 81% to 98.3%.

CHAPTER 7

Conclusions and Future Work

This chapter summarizes the thesis and discusses some interesting directions for future research.

7.1 Thesis Summary

This thesis focuses on the design of Multi-Objective Optimization (MOO) algorithms for the *Cloud Optimizer* in big data systems, addressing: 1) multiple competing performance goals and budgetary costs; 2) high-dimensional parameter space and complex finer-control; and 3) high efficiency for cloud use (i.e., 1-2 seconds). Our experimental evaluation, conducted using MaxCompute in Alibaba Cloud and Spark, demonstrates that our approaches outperform existing methods in both performance and solving time.

More specifically, in Chapter 4, a benchmark analysis of MOO methods and solvers is presented to identify their limitations, particularly regarding efficiency and the quality of Pareto solutions. We compare the performance of three solvers—Knitro, grid search, and random-sampling, and three representative MOO methods: Weighted Sum (WS), Evolutionary (EVO), and Multi-Objective Bayesian Optimization (MOBO). Experimental results demonstrate that the MOBO method and Knitro are not suitable for cloud use due to their high solving time. The other methods are chosen as the MOO baselines for the experiments in Chapters 5 and 6, as they show comparable performance in both quality and efficiency for cloud use.

Chapter 5 proposes a novel hierarchical MOO approach designed to compute

Pareto optimal solutions for query stages in MaxCompute. To fine-tune resources of instances within a stage, this method decomposes the stage-level MOO problem into multiple parallel instance-level MOO problems and efficiently derives stage-level MOO solutions from instance-level MOO solutions. Evaluation results using production workloads demonstrate that our hierarchical MOO approach outperforms existing MOO methods by 4% to 77% in terms of latency performance and up to 48% in cost reduction while operating within 0.02 to 0.24 seconds compared to current optimizers and schedulers.

Chapter 6 develops a new approach, called Hierarchical MOO with Constraints (HMOOC), to achieve Pareto optimality for the entire query with finer-granularity control of Spark parameters. This method decomposes the optimization problem of a large parameter space into smaller subproblems, each constrained to use the same shared parameters. Given that these subproblems are not independent, we develop techniques to generate a sufficiently large set of candidate solutions and efficiently aggregate them to form global Pareto optimal solutions. Evaluation results using TPC-H and TPC-DS benchmarks demonstrate that HMOOC outperforms existing MOO methods, achieving a 4.7% to 54.1% improvement in hypervolume and an 81% to 98.3% reduction in solving time.

7.2 Future Work

For future study, we propose several interesting directions related to this thesis, as outlined below.

Extension to other big data systems. Our MOO modules implemented in MaxCompute and Spark demonstrate superior performance compared to other MOO baselines and the default settings. In the future, we plan to extend our approach to other big data/DBMS systems such as Presto [89] and Greenplum [66].

Take Presto as an example. Presto is a distributed SQL query engine designed for big data environments [89]. It executes queries across a hierarchical structure of stages. To enhance efficiency, Presto incorporates several optimization strategies, including cost-based optimization, history-based optimization, and adaptive

execution. The cost-based optimizer minimizes overall memory usage by optimizing join type selection and join reordering. In its history-based optimizer, Presto leverages precise execution statistics from previously completed queries to improve the performance of future repeated queries. Similarly, akin to Spark's adaptive query execution, Presto enables its planner to re-optimize query plans for downstream tasks during runtime, utilizing statistics gathered from already completed tasks.

Our *Cloud Optimizer* can be integrated with Presto to enhance its performance through the addition of MOO and model server modules. Suppose a user submits a query represented as a Directed Acyclic Graph (DAG) of stages, accompanied by a set of query-level objectives. The MOO module then computes the Pareto optimal solutions, drawing on objective predictions from the model server. For instance, consider fine-grained models, such as stage-level models, where each stage can independently configure its parameters with the objectives of minimizing query-level latency and cost. If the query-level latency and cost are aggregated from stage-level values using `sum` or `max` aggregators, the General Hierarchical MOO method, as proposed in Chapter 5, is appropriate for generating a set of Pareto optimal solutions. Furthermore, if stage-level parameters are subject to constraints, such as sharing certain parameters across all stages, the Hierarchical MOO method with Constraints, discussed in Chapter 6, becomes applicable. In another scenario, where only a coarse-grained model is available that allows for query-level parameter configurations, the MOO module can employ any existing MOO method to directly compute the query-level Pareto optimal solutions.

Workload-level optimization. Query optimization for workload-level optimization in big data systems presents significant challenges due to the complexity of decision-making, influenced by stringent user constraints. For instance, consider a scenario where the objective is to optimize the 99th percentile latency and cost of a workload, all within a one-month budget. In this thesis, our *Cloud Optimizer* takes an analytical task as input, with the MOO module supporting optimization at the task, stage, and query levels by focusing on latency, cost, and other objectives. In practice, performance goals could extend to the workload level, such as optimizing multiple queries within a workload. In the future, the *Cloud Optimizer* could be

extended to support workload-level optimization, making it adaptable to all levels of granularity in optimization.

Recent works addressed workload-level optimization using both coarse-grained and fine-grained approaches [26, 38, 53, 81]. QTune [53] aimed to identify an optimal configuration for entire query workloads to maximize throughput. However, this approach may lead to higher latency as it applies a coarse-grained optimization to the entire query workload rather than tuning individual queries. SparkCruise [81] implemented a feedback loop for Spark workloads, collecting queries to generate a unified workload representation. This facilitates further optimization through the integration of additional optimizer rules. Tuneful [26] prioritized key parameters that influence overall workload performance, rather than focusing on individual queries. Conversely, Microlearner [38] developed a method to segment big data workloads at Microsoft into smaller subsets and created tailored *micromodels* for each subset, enabling more precise optimizations.

The MOO module in the *Cloud Optimizer* is well-suited for addressing workload-level optimization from a fine-grained perspective. For instance, consider a scenario where workload-level latency is represented by the 99th percentile of query-level latencies, and the cost is the aggregate of all query-level costs. Since the 99th percentile latency reflects the maximum latency for the fastest 99% of queries and the cost is computed as the sum of query-level cost, it aligns with the `max` and `sum` aggregation functions employed in the General Hierarchical MOO method. Therefore, when workload-level objectives are aggregated through functions like `max` or `sum` from query-level models, employing the General Hierarchical MOO method within the MOO module is an effective approach for solving this type of multi-objective workload-level optimization.

Enable MOO solutions with model uncertainty. The current MOO module in the *Cloud Optimizer* assumes that predictions from the learned models are accurate. However, the accuracy of these models can significantly impact MOO performance, as inaccurate predictions may lead to incorrect Pareto frontiers.

Future work will focus on integrating model uncertainty into the MOO module to more effectively address and mitigate inaccuracies in the models. There are two

primary existing approaches to solving the MOO problem when model uncertainty is involved. These approaches can generally be classified into: 1) MOO methods, and 2) stochastic methods [60].

The first approach involves quantifying model uncertainty and incorporating it into the objectives, which are then addressed using existing MOO methods. For example, UseMO [9] represents model uncertainty as confidence bounds of predictive models for multiple objectives. Employing a multi-objective Bayesian optimization framework, this method resolves the MOO problem using multiple acquisition functions through NSGA-II [22], selecting new points based on the maximum volume of the uncertainty hyperrectangle.

The second direction transforms the MOO problem with model uncertainty into a stochastic single-objective problem. Stochastic programming methods can tackle this challenge by modeling the randomness in uncertain parameters using probability distributions [13]. This area requires further investigation in future research.

Distributed data centers. Our current cloud optimizer is designed without considering a multi-data center scenario, where an analytical task involves accessing geographically distributed data centers. This introduces additional challenges for MOO, particularly when predictive models are called and updated across distributed locations. In such scenarios, network costs cannot be neglected when invoking predictive models from different locations during optimization.

In the future, our *Cloud Optimizer* could be adapted to effectively handle distributed data centers. Two potential strategies for this adaptation involve enhancements related to the model server and the MOO module within the *Cloud Optimizer*. The first strategy involves integrating data location and network features into the model, enabling the model server to more accurately capture the impact of data distribution and network dynamics. This integration aims to enhance the optimizer's performance by accounting for the complexities of distributed data centers and the variability of network traffic. The second strategy focuses on quantifying network costs and incorporating them into the optimization objectives. These costs could then be addressed using MOO approaches. For instance, NEAL [15] aims to reduce communication time for distributed big data operators. For this strategy, the MOO

module might either utilize existing MOO methods directly if they are deemed suitable, or require the development of novel MOO methods tailored to these specific network challenges. Further research is necessary to explore these directions more comprehensively.

Bibliography

[1] Hani Al-Sayeh, Bunjamin Memishi, et al. "Juggler: Autonomous Cost Optimization and Performance Prediction of Big Data Applications". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Zachary G. Ives, Angela Bonifati, et al. ACM, 2022, pp. 1840–1854. URL: https://doi.org/10.1145/3514221.3517892.

[2] *Amazon EC2 instances*. https://aws.amazon.com/ec2/instance-types.

[3] Michael Armbrust, Reynold S. Xin, et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, et al. ACM, 2015, pp. 1383–1394. URL: https://doi.org/10.1145/2723372.2742797.

[4] Charles Audet, Jean Bigeon, et al. "Performance indicators in multiobjective optimization". In: *European journal of operational research* 292.2 (2021), pp. 397–422.

[5] *Aurora Serverless*. https://aws.amazon.com/rds/aurora/serverless/.

[6] *Automation is the future of cloud cost optimization*. https://www.cncf.io/blog/2021/09/29/automation-is-the-future-of-cloud-cost-optimization/.

[7] Malay Bag, Alekh Jindal, et al. "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*. Ed. by Amar

Phanishayee and Ryan Stutsman. USENIX Association, 2020. URL: https://www.usenix.org/conference/hotcloud20/presentation/bag.

[8] *BARON*. https://minlp.com/baron-solver.

[9] Syrine Belakaria, Aryan Deshwal, et al. "Uncertainty-aware search framework for multi-objective Bayesian optimization". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 06. 2020, pp. 10044–10052.

[10] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of machine learning research* 13.2 (2012).

[11] Ksenia Bestuzheva, Antonia Chmiela, et al. *Global Optimization of Mixed-Integer Nonlinear Programs with SCIP 8*. Tech. rep. 2301.00587. arXiv, 2023.

[12] Muhammad Bilal and Marco Canini. "Towards automatic parameter tuning of stream processing systems". In: *Proceedings of the 2017 Symposium on Cloud Computing*. 2017, pp. 189–200.

[13] JR Birge. *Introduction to stochastic programming*. Springer, 2011.

[14] Ronnie Chaiken, Bob Jenkins, et al. "Scope: easy and efficient parallel processing of massive data sets". In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1265–1276.

[15] Long Cheng, Ying Wang, et al. "Network-aware locality scheduling for distributed data operators in data centers". In: *IEEE Transactions on Parallel and Distributed Systems* 32.6 (2021), pp. 1494–1510.

[16] *CPLEX Optimizer*. https://www.ibm.com/analytics/cplex-optimizer.

[17] *Types of problems solved*. https://www.ibm.com/docs/en/icos/22.1.0?topic=cplex-types-problems-solved.

[18] Claudia D'Ambrosio. "Application-oriented mixed integer non-linear programming". In: *4OR* 8 (2010), pp. 319–322.

[19] Indraneel Das and John E Dennis. "A closer look at drawbacks of minimizing weighted sums of objectives for Pareto set generation in multicriteria optimization problems". In: *Structural optimization* 14.1 (1997), pp. 63–69.

[20] Samuel Daulton, Maximilian Balandat, et al. “Differentiable expected hypervolume improvement for parallel multi-objective Bayesian optimization”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9851–9864.

[21] Kalyanmoy Deb and Himanshu Jain. “An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints”. In: *IEEE transactions on evolutionary computation* 18.4 (2013), pp. 577–601.

[22] Kalyanmoy Deb, Amrit Pratap, et al. “A fast and elitist multiobjective genetic algorithm: NSGA II”. In: *IEEE transactions on evolutionary computation* 6.2 (2002), pp. 182–197.

[23] Michael T. Emmerich and André H. Deutz. “A Tutorial on Multiobjective Optimization: Fundamentals and Evolutionary Methods”. In: *Natural Computing: an international journal* 17.3 (Sept. 2018), pp. 585–609. URL: https://doi.org/10.1007/s11047-018-9685-y.

[24] Martin Ester, Hans-Peter Kriegel, et al. “A density-based algorithm for discovering clusters in large spatial databases with noise”. In: *kdd*. Vol. 96. 34. 1996, pp. 226–231.

[25] Wenchen Fan, Herman van Hovell, et al. *Adaptive Query Execution: Speeding Up Spark SQL at Runtime*. https://www.databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html. 2020.

[26] Ayat Fekry, Lucian Carata, et al. “To Tune or Not to Tune? In Search of Optimal Configurations for Data Analytics”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD ’20. Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 2494–2504. URL: https://doi.org/10.1145/3394486.3403299.

[27] *By 2028, 70% of Workloads Will Run in a Cloud Computing Environment*. https://www.gartner.com/en/conferences/emea/infrastructure-operations-cloud-uk/agenda/day.

[28] *Gartner Says Cloud Will Be the Centerpiece of New Digital Experiences*. https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences.

[29] Michail Georgoulakis Misegiannis, Vasiliki Kantere, et al. "Multi-objective query optimization in Spark SQL". In: *Proceedings of the 26th International Database Engineered Applications Symposium*. 2022, pp. 70–74.

[30] Goetz Graefe. "The Cascades Framework for Query Optimization". In: *IEEE Data Eng. Bull.* 18.3 (1995), pp. 19–29. URL: http://sites.computer.org/debull/95SEP-CD.pdf.

[31] Yacov Y Haimes and David A Wismer. "Integrated system modeling and optimization via quasilinearization". In: *Journal of Optimization Theory and Applications* 8 (1971), pp. 100–109.

[32] Shohedul Hasan, Saravanan Thirumuruganathan, et al. "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*. Ed. by David Maier, Rachel Pottinger, et al. ACM, 2020, pp. 1035–1050. URL: https://doi.org/10.1145/3318464.3389741.

[33] Trevor Hastie, Jerome H. Friedman, et al. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2001. URL: https://doi.org/10.1007/978-0-387-21606-5.

[34] Florian Helff, Le Gruenwald, et al. "Weighted Sum Model for Multi-Objective Query Optimization for Mobile-Cloud Database Environments." In: *EDBT/ICDT Workshops*. Citeseer. 2016, pp. 1–6.

[35] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis Ioannidis. "Schedule Optimization for Data Processing Flows on the Cloud". In: *2011 ACM SIGMOD*. ACM. June 2011, pp. 289–300.

[36] Daniel Hernández-Lobato, Jose Hernandez-Lobato, et al. "Predictive entropy search for multi-objective bayesian optimization". In: *International conference on machine learning*. PMLR. 2016, pp. 1492–1501.

[37] Zhiyao Hu, Dongsheng Li, et al. "Optimizing Resource Allocation for Data-Parallel Jobs Via GCN-Based Prediction". In: *IEEE Trans. Parallel Distributed Syst.* 32.9 (2021), pp. 2188–2201. URL: https://doi.org/10.1109/TPDS.2021.3055019.

[38] Alekh Jindal, Shi Qiao, et al. "Microlearner: A fine-grained learning optimizer for big data workloads at microsoft". In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE. 2021, pp. 2423–2434.

[39] Sangeetha Abdu Jyothi, Carlo Curino, et al. "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*. 2016, pp. 117–134. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/jyothi.

[40] Konstantinos Kanellis, Ramnatthan Alagappan, et al. "Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs". In: *12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*. Ed. by Anirudh Badam and Vijay Chidambaram. USENIX Association, 2020. URL: https://www.usenix.org/conference/hotstorage20/presentation/kanellis.

[41] Florian Karl, Tobias Pielok, et al. "Multi-Objective Hyperparameter Optimization–An Overview". In: *arXiv preprint arXiv:2206.07438* (2022).

[42] *Artelys Knitro User's Manual*. https://www.artelys.com/docs/knitro/index.html.

[43] *Knitro derivatives*. https://www.artelys.com/app/docs/knitro/2_userGuide/derivatives.html.

[44] *Knitro Tips*. https://www.artelys.com/app/docs/knitro/2_userGuide/tips.html.

[45] *Knitro Algorithms (MINLP).* https://www.artelys.com/app/docs/knitro/2_userGuide/minlp.html.

[46] *Knitro Algorithms (NLP).* https://www.artelys.com/app/docs/knitro/2_userGuide/algorithms.html.

[47] Joshua Knowles. "ParEGO: A hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems". In: *IEEE transactions on evolutionary computation* 10.1 (2006), pp. 50–66.

[48] Mina Konakovic Lukovic, Yunsheng Tian, et al. "Diversity-guided multi-objective bayesian optimization with batch evaluations". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 17708–17720.

[49] J Kok Konjaang and Lina Xu. "Multi-objective workflow optimization strategy (MOWOS) for cloud computing". In: *Journal of Cloud Computing* 10.1 (2021), pp. 1–19.

[50] Hsiang-Tsung Kung, Fabrizio Luccio, et al. "On finding the maxima of a set of vectors". In: *Journal of the ACM (JACM)* 22.4 (1975), pp. 469–476.

[51] Mayuresh Kunjir and Shivnath Babu. "Black or White? How to Develop an AutoTuner for Memory-based Analytics". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, et al. ACM, 2020, pp. 1667–1683. URL: https://doi.org/10.1145/3318464.3380591.

[52] Viktor Leis and Maximilian Kuschewski. "Towards cost-optimal query processing in the cloud". In: *Proceedings of the VLDB Endowment* 14.9 (2021), pp. 1606–1612.

[53] Guoliang Li, Xuanhe Zhou, et al. "QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning". In: *Proc. VLDB Endow.* 12.12 (2019), pp. 2118–2130. URL: http://www.vldb.org/pvldb/vol12/p2118-li.pdf.

[54] Jiexing Li, Jeffrey Naughton, et al. "Resource bricolage for parallel database systems". In: *Proceedings of the VLDB Endowment* 8.1 (2014), pp. 25–36.

[55] Yang Li, Huaijun Jiang, et al. “Towards General and Efficient Online Tuning for Spark”. In: *Proc. VLDB Endow.* 16.12 (2023), pp. 3570–3583. URL: https://www.vldb.org/pvldb/vol16/p3570-li.pdf.

[56] Chen Lin, Junqing Zhuang, et al. “Adaptive Code Learning for Spark Configuration Tuning”. In: *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 1995–2007. URL: https://doi.org/10.1109/ICDE53745.2022.00195.

[57] Xi Lin, Zhiyuan Yang, et al. “Pareto set learning for expensive multi-objective optimization”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 19231–19247.

[58] *LINDOGlobal*. https://www.gams.com/latest/docs/S_LINDO.html.

[59] Jie Liu, Wenqian Dong, et al. “Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation”. In: *Proc. VLDB Endow.* 14.11 (2021), pp. 1950–1963. URL: http://www.vldb.org/pvldb/vol14/p1950-liu.pdf.

[60] Suyun Liu and Luis Nunes Vicente. “The stochastic multi-gradient algorithm for multi-objective optimization and its application to supervised machine learning”. In: *Annals of Operations Research* 339.3 (2024), pp. 1119–1148.

[61] Yang Liu, Huanle Xu, et al. “Accordia: Adaptive cloud configuration optimization for recurring data-intensive applications”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 479–479.

[62] Yao Lu, Srikanth Kandula, et al. “Pre-training Summarization Models of Structured Datasets for Cardinality Estimation”. In: *Proc. VLDB Endow.* 15.3 (2021), pp. 414–426. URL: http://www.vldb.org/pvldb/vol15/p414-lu.pdf.

[63] Chenghao Lyu, Qi Fan, et al. *Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing*. Tech. rep. July 2022. URL: https://arxiv.org/pdf/2207.02026.

[64] Chenghao Lyu, Qi Fan, et al. "Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing". In: *Proc. VLDB Endow.* 15.11 (2022), pp. 3098–3111. URL: https://www.vldb.org/pvldb/vol15/p3098-lyu.pdf.

[65] Chenghao Lyu, Qi Fan, et al. *A Spark Optimizer for Adaptive, Fine-Grained Parameter Tuning*. Tech. rep. Mar. 2024. URL: https://chenghao.pages.dev/papers/vldb24-lyu-tr.pdf.

[66] Zhenghua Lyu, Huan Hubert Zhang, et al. "Greenplum: A Hybrid Database for Transactional and Analytical Workloads". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, et al. ACM, 2021, pp. 2530–2542. URL: https://doi.org/10.1145/3448016.3457562.

[67] Ryan Marcus and Olga Papaemmanouil. "WiSeDB: A Learning-based Workload Management Advisor for Cloud Databases". In: *PVLDB* 9.10 (2016), pp. 780–791. URL: http://www.vldb.org/pvldb/vol9/p780-marcus.pdf.

[68] Ryan Marcus and Olga Papaemmanouil. "Plan-Structured Deep Neural Network Models for Query Performance Prediction". In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1733–1746. URL: http://www.vldb.org/pvldb/vol12/p1733-marcus.pdf.

[69] R Timothy Marler and Jasbir S Arora. "Survey of multi objective optimization methods for engineering". In: *Structural and multidisciplinary optimization* 26.6 (2004), pp. 369–395.

[70] *Max Transformation*. https://zhuanlan.zhihu.com/p/65295875.

[71] *Open Data Processing Service*. https://www.alibabacloud.com/product/maxcompute.

[72] Michael D. McKay, Richard J. Beckman, et al. "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output From a Computer Code". In: *Technometrics* 42.1 (2000), pp. 55–61. URL: https://doi.org/10.1080/00401706.2000.10485979.

[73] Achille Messac. “From Dubious Construction of Objective Functions to the Application of Physical Programming”. In: *AIAA Journal* 38.1 (2012), pp. 155–163.

[74] Achille Messac, Amir Ismail-Yahaya, et al. “The normalized normal constraint method for generating the Pareto frontier”. In: *Structural and multidisciplinary optimization* 25.2 (2003), pp. 86–98.

[75] Parimarjan Negi, Ryan C. Marcus, et al. “Flow-Loss: Learning Cardinality Estimates That Matter”. In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2019–2032. URL: http://www.vldb.org/pvldb/vol14/p2019-negi.pdf.

[76] Anish Pimpley, Shuo Li, et al. “Optimal resource allocation for serverless queries”. In: *arXiv preprint arXiv:2107.08594* (2021).

[77] Anish Pimpley, Shuo Li, et al. “Towards Optimal Resource Allocation for Big Data Analytics”. In: *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. Ed. by Julia Stoyanovich, Jens Teubner, et al. OpenProceedings.org, 2022, 2:338–2:350. URL: https://doi.org/10.48786/edbt.2022.20.

[78] *Multiobjective Optimization in Python*. https://platypus.readthedocs.io/en/latest/.

[79] Yuan Qiu, Yilei Wang, et al. “Weighted Distinct Sampling: Cardinality Estimation for SPJ Queries”. In: *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, et al. ACM, 2021, pp. 1465–1477. URL: https://doi.org/10.1145/3448016.3452821.

[80] Kaushik Rajan, Dharmesh Kakadia, et al. “Perforator: eloquent performance models for resource optimization”. In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 2016, pp. 415–427.

[81] Abhishek Roy, Alekh Jindal, et al. “Sparkcruise: Workload optimization in managed spark clusters at microsoft”. In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 3122–3134.

[82] *SCIP Optimization Suite*. https://scipopt.org.

[83] Rathijit Sen, Alekh Jindal, et al. "Autotoken: Predicting peak parallelism for big data analytics at microsoft". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3326–3339.

[84] Rathijit Sen, Abhishek Roy, et al. "AutoExecutor: predictive parallelism for spark SQL queries". In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 2855–2858.

[85] Tarique Siddiqui, Alekh Jindal, et al. "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*. Ed. by David Maier, Rachel Pottinger, et al. ACM, 2020, pp. 99–113. URL: https://doi.org/10.1145/3318464.3380584.

[86] Fei Song, Khaled Zaouk, et al. "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, 2021, pp. 396–407. URL: https://doi.org/10.1109/ICDE51399.2021.00041.

[87] M. van Steen and A.S. Tanenbaum. *Distributed Systems*. 3rd ed. 2017.

[88] Ji Sun, Guoliang Li, et al. "Learned Cardinality Estimation for Similarity Queries". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, et al. ACM, 2021, pp. 1745–1757. URL: https://doi.org/10.1145/3448016.3452790.

[89] Yutian Sun, Tim Meehan, et al. "Presto: A Decade of SQL Analytics at Meta". In: *Proc. ACM Manag. Data* 1.2 (2023), 189:1–189:25. URL: https://doi.org/10.1145/3589769.

[90] Zilong Tan and Shivnath Babu. "Tempo: Robust and Self-Tuning Resource Management in Multi-tenant Parallel Databases". In: *Proc. VLDB Endow.* 9.10 (2016), pp. 720–731. URL: http://www.vldb.org/pvldb/vol9/p720-tan.pdf.

[91] *From Startups To Giants, the Role Of The Cloud In Business Growth*. https://www.forbes.com/sites/jiawertz/2024/02/06/from-startups-to-giants-the-role-of-the-cloud-in-business-growth.

[92] *TPCH Benchmark*. https://www.tpc.org/tpch/default5.asp.

[93] *TPCxBB Benchmark*. http://www.tpc.org/tpcx-bb/.

[94] Immanuel Trummer and Christoph Koch. "Approximation schemes for many-objective query optimization". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 1299–1310.

[95] Immanuel Trummer and Christoph Koch. "An incremental anytime algorithm for multi-objective query optimization". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 1941–1953.

[96] Immanuel Trummer and Christoph Koch. "A fast randomized algorithm for multi-objective query optimization". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 1737–1752.

[97] Dana Van Aken, Andrew Pavlo, et al. "Automatic Database Management System Tuning Through Large-scale Machine Learning". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 1009–1024. URL: http://doi.acm.org/10.1145/3035918.3064029.

[98] Chenxiao Wang, Zach Arani, et al. "Re-optimization for Multi-objective Cloud Database Query Processing using Machine Learning". In: *International Journal of Database Management Systems* 13.1 (2021), pp. 21–40.

[99] Chenxiao Wang, Le Gruenwald, et al. "Cloud Query Processing with Reinforcement Learning-Based Multi-objective Re-optimization". In: *Model and Data Engineering - 10th International Conference, MEDI 2021, Tallinn, Estonia, June 21-23, 2021, Proceedings*. Ed. by J. Christian Attiogbé and Sadok Ben Yahia. Vol. 12732. Lecture Notes in Computer Science. Springer, 2021, pp. 141–155. URL: https://doi.org/10.1007/978-3-030-78428-7%5C_12.

[100] Chenxiao Wang, Le Gruenwald, et al. “SLA-Aware Cloud Query Processing with Reinforcement Learning-based Multi-Objective Re-Optimization”. In: *International Conference on Big Data Analytics and Knowledge Discovery*. Springer. 2022, pp. 249–255.

[101] Jiayi Wang, Chengliang Chai, et al. “FACE: A Normalizing Flow based Cardinality Estimator”. In: *Proc. VLDB Endow.* 15.1 (2021), pp. 72–84. URL: http://www.vldb.org/pvldb/vol15/p72-li.pdf.

[102] Junxiong Wang, Immanuel Trummer, et al. “UDO: Universal Database Optimization Using Reinforcement Learning”. In: *Proc. VLDB Endow.* 14.13 (Sept. 2021), pp. 3402–3414. URL: https://doi.org/10.14778/3484224.3484236.

[103] *What does CPLEX do?* https://www.ibm.com/docs/en/icos/22.1.1?topic=cplex-what-does-do.

[104] James Wilson, Frank Hutter, et al. “Maximizing acquisition functions for Bayesian optimization”. In: *Advances in Neural Information Processing Systems* 31 (2018), pp. 9884–9895.

[105] Lucas Woltmann, Dominik Olwig, et al. “PostCENN: PostgreSQL with Machine Learning Models for Cardinality Estimation”. In: *Proc. VLDB Endow.* 14.12 (2021), pp. 2715–2718. URL: http://www.vldb.org/pvldb/vol14/p2715-woltmann.pdf.

[106] Peizhi Wu and Gao Cong. “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*. Ed. by Guoliang Li, Zhanhuai Li, et al. ACM, 2021, pp. 2009–2022. URL: https://doi.org/10.1145/3448016.3452830.

[107] Ziniu Wu, Amir Shaikhha, et al. *BayesCard: Revitilizing Bayesian Frameworks for Cardinality Estimation*. 2020. URL: https://arxiv.org/abs/2012.14743.

[108] Jinhan Xin, Kai Hwang, et al. “LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications”. In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. Ed. by Zachary G. Ives, Angela Bonifati, et al. ACM, 2022, pp. 674–684. URL: https://doi.org/10.1145/3514221.3526157.

[109] Matei Zaharia, Mosharaf Chowdhury, et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. Ed. by Steven D. Gribble and Dina Katabi. USENIX Association, 2012, pp. 15–28. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[110] Ji Zhang, Yu Liu, et al. “An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning”. In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: ACM, 2019, pp. 415–432. URL: http://doi.acm.org/10.1145/3299869.3300085.

[111] Qingfu Zhang and Hui Li. “MOEA/D: A multiobjective evolutionary algorithm based on decomposition”. In: *IEEE Transactions on evolutionary computation* 11.6 (2007), pp. 712–731.

[112] Xinyi Zhang, Zhuo Chang, et al. “A Unified and Efficient Coordinating Framework for Autonomous DBMS Tuning”. In: *CoRR* abs/2303.05710 (2023). arXiv: 2303.05710. URL: https://doi.org/10.48550/arXiv.2303.05710.

[113] Xinyi Zhang, Hong Wu, et al. “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*. Ed. by Guoliang Li, Zhanhuai Li, et al. ACM, 2021, pp. 2102–2114. URL: https://doi.org/10.1145/3448016.3457291.

[114] Xinyi Zhang, Hong Wu, et al. “Towards dynamic and safe configuration tuning for cloud databases”. In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 631–645.

[115] Zhuo Zhang, Chao Li, et al. “Fuxi: a Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale”. In: *Proc. VLDB Endow.* 7.13 (2014), pp. 1393–1404. URL: http://www.vldb.org/pvldb/vol7/p1393-zhang.pdf.

[116] Yiyang Zhao, Linnan Wang, et al. “Multi-objective Optimization by Learning Space Partition”. In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: https://openreview.net/forum?id=FlwzVjfMryn.

[117] Rong Zhu, Ziniu Wu, et al. “FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation”. In: *Proc. VLDB Endow.* 14.9 (2021), pp. 1489–1502. URL: http://www.vldb.org/pvldb/vol14/p1489-zhu.pdf.

[118] Yuqing Zhu and Jianxun Liu. “ClassyTune: A Performance Auto-Tuner for Systems in the Cloud”. In: *IEEE Trans. Cloud Comput.* 10.1 (2022), pp. 234–246. URL: https://doi.org/10.1109/TCC.2019.2936567.

[119] Yuqing Zhu, Jianxun Liu, et al. “Bestconfig: tapping the performance potential of systems via automatic configuration tuning”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. 2017, pp. 338–350.

[120] Eckart Zitzler, Marco Laumanns, et al. “SPEA2: Improving the strength Pareto evolutionary algorithm”. In: *TIK report* 103 (2001).

APPENDIX A

Additional Materials for Benchmark Study of Multi-Objective Optimization

Unlike Knitro [42], CPLEX primarily functions as a Linear Programming (LP) solver [16]. We provide a theoretical analysis on the feasibility of transforming ReLU activations into linear expressions and extend this transformation approach to the broader context of MINLP problems. This analysis seeks to explore the potential for representing Mixed Integer Non-Linear Programming (MINLP) problems within a linear framework.

A.1 CPLEX

Consider CPLEX [16] as an example of an LP solver. CPLEX is an optimization software package designed for solving LP problems. The representation of the optimization problem solved by CPLEX follows the format defined in Equation (4.4). However, CPLEX is limited to LP and its extensions, such as Quadratic Programming (QP), Mixed Integer Programming (MIP), and network-flow problems [17, 103]. The general implementation in CPLEX involves configuring properties for objectives, constraints, and variables.

Theoretically, CPLEX can solve our MINLP problem once the non-linear representation of the DNN is converted into a linear expression. Specifically, the crucial step is transforming the ReLU function into a linear representation. After this trans-

formation, the MINLP optimization problem falls into the MIP category due to the mixed variable types. CPLEX addresses this using a highly general and robust algorithm based on branch-and-cut.

ReLU transformation. The main idea is to convert the maximization function into a constrained linear representation by introducing new variables, as discussed in an online math community [70]. For example, consider the ReLU function $y = \max(x_1w_1 + x_2w_2, 0)$. The conversion steps are as follows:

1. Add a new variable x_0 and let $x_0 = \max(x_1w_1 + x_2w_2, 0)$;

2. Add a new binary variable δ and then convert the max function into linear constraints using the formulation in Equation (A.1), where M represents a sufficiently large constant (big-M).

$$\text{subject to} \begin{cases} x_0 \leq x_1w_1 + x_2w_2 + M(1-\delta) \\ x_0 \geq x_1w_1 + x_2w_2 - M(1-\delta) \\ x_0 \leq 0 + M\delta \\ x_0 \geq 0 - M\delta \\ x_1w_1 + x_2w_2 \leq 0 + M\delta \\ 0 \leq x_1w_1 + x_2w_2 + M(1-\delta) \end{cases} \tag{A.1}$$

Finally, Equation (4.5) is reformulated to minimize $y = x_0$ with constraints defined in Formula (A.1), by introducing one new variable, one new binary variable, and six corresponding constraints.

To demonstrate the validity of this transformation, note that since δ is a binary variable included in the constraints, the following is evident: 1) when $\delta = 0$, it results in $x_0 = 0$; 2) when $\delta = 1$, it results in $x_0 = x_1w_1 + x_2w_2$.

MINLP transformation. We now extend the ReLU transformation to a more general MINLP transformation. After this conversion, the MINLP problem is reformulated as a constrained MIP problem, where both the objective function and constraints are linear.

Starting with the two-layer DNN defined in Equation (4.2), the corresponding representation of the objective, as shown in Equations (4.6) and (4.7), includes three ReLU functions. Applying the conversion steps, the MINLP is reformulated to include three new variables, three new binary variables, and eighteen constraints.

Finally, the MINLP problem is converted into a constrained MIP problem, aiming to minimize $y = x_0^{'}$, where

$$x_0^{'} = \max(x_0^{11} w_1^2 + x_0^{12} w_2^2, 0) \tag{A.2}$$

with x_0^{11} and x_0^{12} represented as:

$$\begin{aligned} x_0^{11} &= \max(x_1 w_{11}^1 + x_2 w_{12}^1, 0) \\ x_0^{12} &= \max(x_1 w_{21}^1 + x_2 w_{22}^1, 0) \end{aligned} \tag{A.3}$$

Additional new binary variables and constraints are introduced for $x_0^{11}, x_0^{12}, x_0^{'}$, following the formulation in Equation (A.1).

Overall, the MINLP transformation introduces new variables and constraints, with their total numbers dependent on the DNN configuration. For a MINLP with an arbitrary DNN model as the objective, where the DNN includes L layers, m nodes in each hidden layer, and one node in the output layer, the total number of ReLU functions is $m \times (L-1) + 1$. For each ReLU function, two new variables and six constraints are introduced to convert the MINLP to a MIP. Consequently, the total number of new variables is $2 \times (m \times (L-1) + 1)$, and the total number of new constraints is $6 \times (m \times (L-1) + 1)$. This can lead to a large-scale optimization problem when L and m are large.

ECOLE
DOCTORALE

Titre : Optimisation multi-objectifs pour l'analyse de données dans le cloud

Mots clés : Multi-objectif, optimisation, cloud computing, réglage des paramètres, base de données

Résumé : Le traitement des requêtes Big Data est devenu de plus en plus important, ce qui a conduit au développement et au déploiement dans le cloud de nombreux systèmes. Cependant, le réglage automatique des nombreux paramètres de ces systèmes Big Data introduit une complexité croissante dans la satisfaction des objectifs de performance et des contraintes budgétaires des utilisateurs. Déterminer les configurations optimales est un défi en raison de la nécessité de répondre à : 1) de multiples objectifs de performance concurrents et des contraintes budgétaires, tels qu'une faible latence et un faible coût, 2) un espace de paramètres de grande dimension avec un contrôle de paramètres complexe, et 3) l'exigence d'une efficacité de calcul élevée dans l'utilisation du cloud, généralement dans les 1 à 2 secondes.
Pour relever les défis ci-dessus, cette thèse propose des algorithmes d'optimisation multi-objectifs (MOO) efficaces pour un optimiseur de cloud afin de répondre à divers objectifs utilisateur. Il calcule des configurations optimales de Pareto pour les requêtes Big Data dans un espace de paramètres de grande dimension tout en respectant des exigences strictes en matière de temps de résolution. La première contribution de cette thèse est une analyse comparative des méthodes et des solveurs MOO existants, identifiant leurs limites lorsqu'ils sont appliqués à l'optimisation du cloud. La deuxième contribution présente les algorithmes MOO conçus pour calculer des solutions optimales de Pareto pour les étapes de requête, qui sont des unités définies par des limites de mélange. Notre troisième contribution vise à atteindre l'optimalité de Pareto pour l'ensemble de la requête avec un contrôle de granularité plus fin des paramètres.
Les résultats d'évaluation basés sur des systèmes de big data existants démontrent que nos approches surpassent considérablement les algorithmes MOO existants en termes de mesures de performance telles que les réductions simultanées de la latence et du coût, ou les améliorations de l'hypervolume, le tout obtenu dans un temps de résolution faible (moins de 2 secondes).

Title : Multi-Objective Optimization for Data Analytics in the Cloud

Keywords : Multi-objective, optimization, cloud computing, parameter tuning, database

Abstract : Big data query processing has become increasingly important, prompting the development and cloud deployment of numerous systems. However, automatically tuning the numerous parameters in these big data systems introduces growing complexity in meeting users' performance goals and budgetary constraints. Determining optimal configurations is challenging due to the need to address : 1) multiple competing performance goals and budgetary constraints, such as low latency and low cost, 2) a high-dimensional parameter space with complex parameter control, and 3) the requirement for high computational efficiency in cloud use, typically within 1-2 seconds.
To address the above challenges, this thesis proposes efficient multi-objective optimization (MOO) algorithms for a cloud optimizer to meet various user objectives. It computes Pareto optimal configurations for big data queries within a high-dimensional parameter space while adhering to stringent solving time requirements. The first contribution of this thesis is a benchmarking analysis of existing MOO methods and solvers, identifying their limitations when applied to cloud optimization. The second contribution introduces MOO algorithms designed to compute Pareto optimal solutions for query stages, which are units defined by shuffle boundaries. Our third contribution aims to achieve Pareto optimality for the entire query with finer-granularity control of parameters.
Evaluation results based on existing big data systems demonstrate that our approaches significantly outperform existing MOO algorithms in terms of performance metrics such as simultaneous reductions in latency and cost, or improvements in hypervolume, all achieved within a low solving time (less than 2 seconds).

Institut Polytechnique de Paris
91120 Palaiseau, France