A Spark Optimizer for Adaptive, Fine-Grained Parameter Tuning

Chenghao Lyu
University of Massachusetts, Amherst
chenghao@cs.umass.edu

Qi Fan, Philippe Guyard
Ecole Polytechnique
qi.fan@polytechnique.edu
philippe.guyard@polytechnique.edu

Yanlei Diao
Ecole Polytechnique
University of Massachusetts, Amherst
yanlei.diao@polytechnique.edu

ABSTRACT

As Spark becomes a common big data analytics platform, its growing complexity makes automatic tuning of numerous parameters critical for performance. Our work on Spark parameter tuning is particularly motivated by two recent trends: Spark's *Adaptive Query Execution* (AQE) based on runtime statistics, and the increasingly popular *Spark cloud deployments* that make cost-performance reasoning crucial for the end user. This paper presents our design of *a Spark optimizer that controls all tunable parameters of each query in the new AQE architecture to explore its performance benefits and, at the same time, casts the tuning problem in the theoretically sound multi-objective optimization (MOO) setting to better adapt to user cost-performance preferences.* To this end, we propose a novel hybrid compile-time/runtime approach to multi-granularity tuning of diverse, correlated Spark parameters, as well as a suite of modeling and optimization techniques to solve the tuning problem in the MOO setting while meeting the stringent time constraint of 1-2 seconds for cloud use. Evaluation results using TPC-H and TPC-DS benchmarks demonstrate the superior performance of our approach: (*i*) When prioritizing latency, it achieves 63% and 65% reduction for TPC-H and TPC-DS, respectively, under an average solving time of 0.7-0.8 sec, outperforming the most competitive MOO method that reduces only 18-25% latency with 2.6-15 sec solving time. (*ii*) When shifting preferences between latency and cost, our approach dominates the solutions of alternative methods, exhibiting superior adaptability to varying preferences.

PVLDB Reference Format:
Chenghao Lyu, Qi Fan, Philippe Guyard, and Yanlei Diao. A Spark Optimizer for Adaptive, Fine-Grained Parameter Tuning. PVLDB, 17(11): 3565-3579, 2024.
doi:10.14778/3681954.3682021

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/udao-moo/udao-spark-optimizer.

1 INTRODUCTION

Big data query processing has become an integral part of enterprise businesses and many platforms have been developed for this purpose [3, 6, 7, 12, 15, 36, 41, 51, 56, 63, 64, 69, 70]. As these systems are becoming increasingly complex, parameter tuning of big data systems has recently attracted a lot of research attention [23, 25, 27, 28, 47, 62]. Take Apache Spark for example. It offers over 180 parameters for governing a *mixed set of decisions*, including resource allocation, the degree of parallelism, shuffling behaviors, and SQL-related decisions. Our work on parameter tuning of big data query systems is particularly motivated by two recent trends:

Adaptive Query Execution. Big data query processing systems have undergone architectural changes that distinguish them substantially from traditional DBMSs for the task of parameter tuning. A notable feature is that a SQL query is compiled into a physical plan composed of query stages and a query stage is the granularity of scheduling and execution. The stage-based query execution model enables the system to observe the precise statistics of the completed stages at runtime. Recently, Spark has taken a step further to introduce Adaptive Query Execution (AQE), which upon the completion of each query stage, considers runtime statistics and re-optimizes the logical query plan to a new physical plan using parametric rules. Spark, however, does not support parameter tuning itself and instead, executes AQE based on the default or pre-specified configuration of the parameters. Hence, it can suffer from suboptimal performance of AQE when the parameters are not set to appropriate values. On the other hand, recent work on Spark tuning [23, 25, 27, 28, 47, 62] has limited itself to the traditional setting that the parameters are set at query submission time and then fixed throughout query execution, hence missing the opportunity of exploring AQE to improve the physical query plan.

Cost-performance reasoning in cloud deployment. As big data query processing is increasingly deployed in the cloud, parameter tuning in the form of cost-performance optimization [32, 42] has become more critical than ever to end users. Prior work [25, 66, 73] has used fixed weights to combine multiple objectives into a *single objective* (SO) and solve the SO problem to return one solution. However, the optimization community has established theory [35] pointing out that solving such a SO problem is unlikely to return a solution that balances the cost-performance as the specified weights intend to express (as we will demonstrate in this work). The theoretically sound approach to adapting between cost and performance is to treat it as a *multi-objective optimization* (MOO) problem [9, 35, 38, 39], compute the Pareto optimal set, and return one solution from the set that best matches the user preference as reflected by the weights set on the objectives [32, 47].

Therefore, our work in this paper aims to *design a Spark optimizer that controls all tunable parameters (collectively called a "configuration") of each Spark application in the new architecture of adaptive query execution to explore its performance benefits and, at the same time, casts the tuning problem in the multi-objective optimization setting to better adapt to user cost-performance needs.* This Optimizer for Parameter Tuning (OPT) complements Spark's cost and rule-based optimization of query plans, where the optimization rules

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682021

use default or pre-specified values of Spark parameters. Our OPT can be implemented as a plugin in the current Spark optimizer framework and runs each time a query is submitted for execution.

Designing the optimizer for parameter tuning, as defined above, faces a few salient challenges:

Complex control of a mixed parameter space. One may wonder whether parameter tuning can be conducted solely at runtime, as an augmented AQE process. Unfortunately, Spark parameter tuning is more complex than that due to the need to control a mixed parameter space. More specifically, Spark parameters can be divided into three categories (see Table 1 for examples): the context parameters, θ_c, initialize the Spark context by specifying (shared) resources to be allocated and later controlling various runtime behaviors; the query plan parameters, θ_p, govern the translation from the logical to physical query plan; and the query stage parameters, θ_s, govern the optimization of the query stages in the physical plan. The θ_p and θ_s parameters are best tuned at runtime to benefit from precise statistics, but they are strongly correlated with the context parameters, θ_c, which control shared resources and must be set at query submission time to initialize the Spark context. How to best tune these mixed parameters, correlated but under different controls in the query lifetime, is a nontrivial issue.

Stringent MOO Solving time for cloud use. The second challenge is solving the MOO problem over a large parameter space of Spark while obeying stringent time constraints for cloud use, e.g., under 1-2 seconds to avoid delays in starting a Spark application in serverless computing [1, 32, 46]. Prior work on MOO for Spark tuning [47] has reported the running time of the Evolutional method [9] to be about 5 seconds for query-level control of 12 Spark parameters. In our work, when we allow the θ_p parameters to be tuned separately for different subqueries, the time cost of Evolutional increases beyond 60 seconds for some TPC-H queries, which is unacceptable for cloud use.

To address the above challenges, we propose a novel approach to multi-granularity tuning of mixed Spark parameters and a suite of modeling and optimization techniques to solve the tuning problem in the MOO setting efficiently and effectively. More specifically, our contributions include the following:

1. A hybrid approach to OPT (Section 3): Our OPT is designed for multi-granularity tuning of mixed parameters: while the context parameters θ_c configure the Spark context at the *query level*, we tune the θ_p and θ_s parameters at the fine-grained *subquery level* and *query stage level*, respectively, to maximize performance gains. To cope with Spark's different control mechanisms for these parameters, we introduce a new hybrid compile-time/runtime optimization approach to multi-granularity tuning: the compile-time optimization finds the optimal θ_c^*, by leveraging the correlation between θ_c and fine-grained $\{\theta_p\}$ and $\{\theta_s\}$, to construct an ideal Spark context for query execution. Then the runtime optimization adjusts fine-grained $\{\theta_p\}$ and $\{\theta_s\}$ based on the precise statistics of the completed stages. Both compile-time and runtime optimization are cast in the setting of multi-objective optimization.

2. Modeling (Section 4): Solving the MOO problem for parameter tuning requires precise models for the objective functions used. Our hybrid approach requires accurate models for both compile-time and runtime optimization, which use different representations of query plans. The Spark execution environment shares resources among parallel stages, which further complicates the modeling problem. To address all of these issues, we introduce a modeling framework that combines a Graph Transformer Network (GTN) embedder of query plans and a regression model that captures the interplay of the tunable parameters (decision variables) and critical contextual factors (non-decision variables) such as query and data characteristics and resource contention. We devise new techniques to construct compile-time and runtime models in this framework.

3. MOO Algorithms (Section 5): Solving the MOO problem for multi-granularity tuning needs to conquer the high-dimensionality of the parameter space while obeying the time constraint, which is especially the case at compile-time when we consider the correlation of all the parameters together. We introduce a novel approach for compile-time optimization, named Hierarchical MOO with Constraints (HMOOC): it breaks the optimization problem of a large parameter space into a set of smaller problems, one for each subquery, but subject to the constraint that all subquery-level problems use the same Spark context parameters, θ_c. Since these subproblems are not independent, we devise a host of techniques to prepare a sufficiently large set of candidate solutions for the subproblems and efficiently aggregate them to build global Pareto optimal solutions. Then our runtime optimization runs as part of AQE to adapt θ_p and θ_s effectively based on precise statistics.

Evaluation results using TPC-H and TPC-DS benchmarks demonstrate the superior performance of our techniques. (1) Our compile-time MOO algorithm for fine-grained parameter tuning outperforms existing MOO methods with 4.7%-54.1% improvement in hypervolume (the dominated space by the Pareto front) and 81%-98.3% reduction in solving time. (2) Our compile-time/runtime optimization, when asked to prioritize latency, achieves 63% and 65% reduction for TPC-H and TPC-DS, respectively, under an average solving time of 0.7-0.8 sec, whereas the most competitive MOO method reduces only 18-25% latency with high solving time of 2.6-15 sec. When shifting preferences between latency and cost, our approach dominates the solutions of alternative methods by a wide margin, exhibiting superior adaptability to varying preferences.

2 RELATED WORK

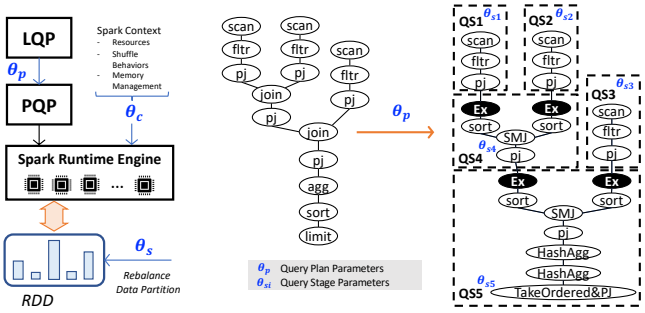
DBMS tuning. Our problem is related to a body of work on performance tuning for DBMSs. Most DBMS tuning systems employ an *offline*, iterative tuning session for each workload [55, 58, 66, 67], which can take long to run (e.g., 15-45 minutes [55, 66]). Ottertune [55] builds a predictive model for each query by leveraging similarities to past queries, and runs Gaussian Process (GP) exploration to try other configurations to reduce query latency. ResTune [67] accelerates the GP tuning process (with cubic complexity in the number of training samples) by building a meta-learning model to weigh appropriately the base learners trained for individual tuning tasks. CDBTune [66] and QTune [25] use Deep Reinforcement Learning (RL) to predict the reward of a configuration, as the weighted sum of different objectives, and explores new configurations to optimize the reward. These methods can take many iterations to achieve good performance [62]. UDO [58] is an offline RL-based tuner for both database physical design choices and parameter settings. OnlineTuner [68] tunes workloads in the online setting by exploring a contextual GP to adapt to changing contexts

and safe exploration strategies. Our work on parameter tuning aims to be part of the Spark optimizer, invoked on-demand for each arriving query, hence different from all the tuning systems that require launching a separate tuning session for each target workload.

Tuning of big data systems. Among *search-based* methods, BestConfig [73] searches for good configurations by dividing high-dimensional configuration space into subspaces based on samples, but it cold-starts each tuning request. ClassyTune [72] solves the optimization problem by classification, and Li et al. [26] prunes searching space with a running environment independent cost model, both of which cannot be easily extended to the MOO setting. A new line of work considered Spark parameter tuning for recurring workloads. ReIM [23] tunes memory management decisions online by guiding the GP approach using manually-derived memory models. Locat [62] is a data-aware GP approach for tuning Spark queries that repeatedly run with the input data size changing over time. While it outperforms prior solutions such as Tuneful [11], ReIM [23], and QTune [25] in efficiency, it still needs hours to complete. Li et al. [27] further tune periodic Spark jobs using a GP with safe regions and meta-learning from history. LITE[28] tunes parameters of non-SQL Spark applications and relies on stage code analysis to derive predictive models, which is impractical as cloud providers usually have no access to application code under privacy constraints. These solutions do not suit our problem as we cannot afford to launch a separate tuning session for each query or target workload, and these methods lack support of adaptive runtime optimization and are limited to single-objective optimization.

Resource optimization in big data systems. In cluster computing, a resource optimizer (RO) determines the optimal resource configuration *on demand* and *with low latency* as jobs are submitted. Morpheus [18] codifies user expectations as multiple Service-Level Objectives (SLOs) and enforces them using scheduling methods. However, its optimization focuses on system utilization and predictability, but not cost and latency of Spark queries. PerfOrator [45] optimizes latency via an exhaustive search of the solution space while calling its model for predicting the performance of each solution. WiseDB [34] manages cloud resources based on a decision tree trained on minimum-cost schedules of sample workloads. ReLocag[16] presents a predictor to find the near-optimal number of CPU cores to minimize job completion time. Recent work [24] proposes a heuristic-based model to recommend a cloud instance that achieves cost optimality for OLAP queries. This line of work addresses a smaller set of tunable parameters (only for resource allocation) than the general problem of Spark tuning with large parameter space, and is limited to single-objective optimization.

Multi-objective optimization (MOO) computes a set of solutions that are not dominated by any other configuration in all objectives, aka, the Pareto-optimal set (or Pareto front). Theoretical MOO solutions suffer from various performance issues in cloud optimization: Weighted Sum [35] is known to have *poor coverage* of the Pareto front [38]. Normalized Constraints [39] lacks in *efficiency* due to repeated recomputation to return more solutions. Evolutionary methods [9] approximately compute a Pareto set but suffer from *inconsistent solutions*. Multi-objective Bayesian Optimization [5, 14] extends the Bayesian approach to modeling an unknown function with an acquisition function for choosing the next point(s) that are



(a) Mixed control in query lifetime

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

Figure 1: Spark parameters provide mixed control through query compilation and execution

likely to be Pareto optimal. But it is shown to take long to run [47] and hence lacks the *efficiency* required by a cloud optimizer.

In the DB literature, MOO for SQL queries [17, 20, 52–54] finds Pareto-optimal query plans by efficiently searching through a large set of plans. The problem, essentially a combinatorial one, differs from MOO for parameter tuning, which is a numerical optimization problem. TEMPO [50] considers multiple SLOs of SQL queries and guarantees max-min fairness when they cannot be all met. MOO for workflow scheduling [20] assigns operators to containers to minimize total running time and money cost, but is limited to searching through 20 possible containers and solving a constrained optimization for each option.

The closest work to ours is UDAO [47, 65] that tunes Spark configurations to optimize for multiple objectives. It *Progressive Frontier* (PF) method [47] provides the MOO solution for spark parameter tuning with good coverage, efficiency, and consistency. However, the solution is limited to coarse-grained query-level control of parameters. Lyu et al. extended the MOO solution to serverless computing [32] by controlling machine placement and resource allocation to parallel tasks of each query stage. However, its solution only guarantees Pareto optimality for each individual stage, but not the entire query (with potentially many stages).

3 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.

3.1 Background on Spark

Apache Spark [64] 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 [2]. A submitted SQL query undergoes parsing,

Table 1: Example Spark parameters in three categories

θ_c	**Context Parameters** (`spark.*`)
k_1	`executor.cores`
k_2	`executor.memory`
k_3	`executor.instances`
θ_p	**Logical Query Plan Parameters** (`spark.sql.*`)
s_1	`adaptive.advisoryPartitionSizeInBytes`
s_3	`adaptive.maxShuffledHashJoinLocalMapThreshold`
s_4	`adaptive.autoBroadcastJoinThreshold`
s_5	`shuffle.partitions`
θ_s	**Query Stage Parameters** (`spark.sql.adaptive.*`)
s_{10}	`rebalancePartitionsSmallPartitionFactor`

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.

The execution of a Spark SQL query is configured by three categories of parameters, as shown in Table 1, providing different controls in query lifetime. As Figure 1(a) shows, **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 1(b) shows a concrete example of translating a LQP to PQP, 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 management 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 [13, 29, 30, 43, 44, 48, 57, 59–61, 71] 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 [10]. Figure 2 shows the life cycle of a SQL query with the AQE mechanism turned on. 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

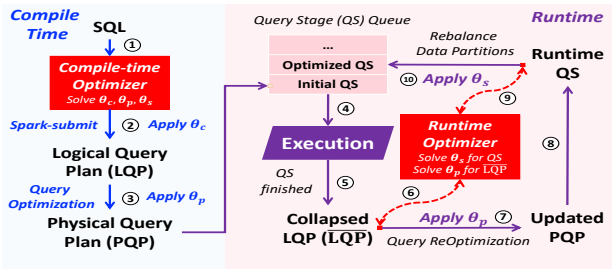


Figure 2: Query life cycle with an optimizer for parameter tuning

rule is configured by a subset of $\boldsymbol{\theta_p}$ or $\boldsymbol{\theta_s}$ parameters. The details of those rules are left to [31].

3.2 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 3(a) shows that for TPCH-Q9, query-level parameter tuning using a prior MOO method [47] and then running AQE (the middle bar) can already provide a 13% improvement over AQE with the default configuration (left bar).

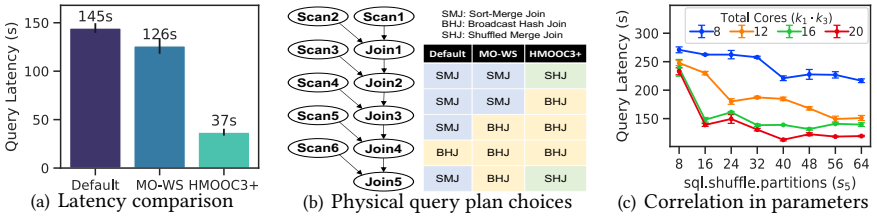
Second, *fine-grained tuning has performance benefits over query-level tuning.* While existing work on Spark parameter tuning [23, 25, 27, 28, 47, 62] focuses on query-level tuning, we show in Figure 3(a) that adapting $\boldsymbol{\theta_p}$ for different collapsed query plans during runtime can further reduce the latency by 61% (the right bar). Figure 3(b) shows the simplified query structure of TPCH-Q9, including 6 scan operators and 5 join operators. 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 (SMJ) + 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 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 $\boldsymbol{\theta_c}$. Many similar examples exist.

3.3 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.

3.3.1 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



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

Figure 3: Profiling TPCH-Q9 (12 subQs) over different configurations

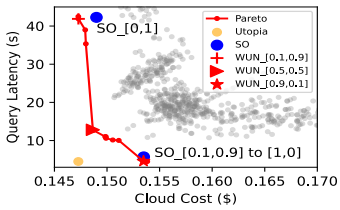


Figure 4: MOO solutions for TPCH Q2

$\boldsymbol{\theta}_p$ 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 2.

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

3.3.2 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.

Prior work [25, 66, 73] 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) [35], 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 (i.e., $n = 1$ in `SO-FW`). 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 [35]. Empirically, `MO-WS` has been reported with sparse coverage of the Pareto front in prior work [47]. Figure 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. 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, our work casts the optimization problem in the *multi-objective optimization* (MOO) framework [9, 35, 38, 39], which computes the Pareto front properly to capture the tradeoffs and later allows us to recommend one solution that best matches the user preference.

Formally, a MOO problem aims to minimize multiple objectives simultaneously, where the objectives are represented as functions $\boldsymbol{f} = (f_1, ..., f_k)$ on all the tunable parameters $\boldsymbol{\theta}$.

Definition 3.1. Multi-Objective Optimization (MOO).

$$\arg\min_{\boldsymbol{\theta}} \quad \boldsymbol{f}(\boldsymbol{\theta}) = [f_1(\boldsymbol{\theta}), f_2(\boldsymbol{\theta}), \ldots, f_k(\boldsymbol{\theta})]$$

$$s.t. \quad \begin{aligned} &\boldsymbol{\theta} \in \Sigma \subseteq \mathbb{R}^d \\ &\boldsymbol{f}(\boldsymbol{\theta}) \in \Phi \subseteq \mathbb{R}^k \\ &L_i \le f_i(\boldsymbol{\theta}) \le U_i, \quad i = 1, ..., k \end{aligned}$$

where $\boldsymbol{\theta}$ is the configuration with d parameters, $\Sigma \subseteq \mathbb{R}^d$ denotes all possible configurations, and $\Phi \subseteq \mathbb{R}^k$ denotes the objective space. If an objective favors larger values, we add the minus sign to the objective function to transform it into a minimization problem.

Definition 3.2. Pareto Optimal Set. In the objective space $\Phi \subseteq \mathbb{R}^k$, a point $\boldsymbol{F'}$ Pareto-dominates another point $\boldsymbol{F''}$ iff $\forall i \in [1, k], F'_i \le F''_i$ and $\exists j \in [1, k], F'_j < F''_j$. For a given query, solving the MOO problem leads to a Pareto Set (Front) $\mathcal{F}$ that includes all the Pareto optimal solutions $\{(\boldsymbol{F}, \boldsymbol{\theta})\}$, where $\boldsymbol{F}$ is a Pareto point in the objective space Φ and $\boldsymbol{\theta}$ is its corresponding configuration in Σ.

Figure 4 shows a Pareto front for TPCH-Q2. Most configurations, depicted by the grey dots, are dominated by the Pareto optimal configurations, depicted by the 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 [47] 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 4 shows a few recommendations when we run WUN on the Pareto front with different weighted preferences on the objectives.

We next define the MOO problem for Spark parameter tuning.

Definition 3.3. Multi-Objective Optimization for Spark SQL

$$\arg\min_{\boldsymbol{\theta}_c, \{\boldsymbol{\theta}_p\}, \{\boldsymbol{\theta}_s\}} \boldsymbol{f}(\boldsymbol{\theta}_c, \{\boldsymbol{\theta}_p\}, \{\boldsymbol{\theta}_s\}) = \begin{bmatrix} f_1(\text{LQP}, \boldsymbol{\theta}_c, \{\boldsymbol{\theta}_p\}, \{\boldsymbol{\theta}_s\}, \alpha, \beta, \gamma) \\ \ldots \\ f_k(\text{LQP}, \boldsymbol{\theta}_c, \{\boldsymbol{\theta}_p\}, \{\boldsymbol{\theta}_s\}, \alpha, \beta, \gamma) \end{bmatrix}$$

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

Table 2: Comparison of Spark parameter tuning methods

	Mixed Param. Space	Adaptive Runtime Opt.	Multi-Granularity	Multi-Objective
ReLocag [16]	×	×	×	×
BestConfig [73]	✓	×	×	×
ClassyTune [72]	✓	×	×	×
LITE [28]	✓	×	×	×
LOCAT [62]	✓	×	×	×
Li et. al [27]	✓	×	×	×
UDAO [47]	✓	×	×	✓
Ours	✓	✓	✓	✓

where LQP denotes the logical query plan with operator cardinality estimates, and θ_c, $\{\theta_p\}$, $\{\theta_s\}$ represent the *decision variables* configuring Spark context, LQP transformations, and query stage (QS) optimizations, respectively. More specifically, $\{\theta_p\}$ is the collection of all LQP parameters, and θ_{pt} is a copy of θ_p for the t-th transformation of the collapsed query plan $\overline{\text{LQP}}$. Similarly, $\{\theta_s\}$ is the collection of QS parameters, and θ_{si} is a copy for optimizing query stage i. $\Sigma_c, \Sigma_p, \Sigma_s$ are the feasible space for θ_c, θ_p and θ_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.

3.3.3 Comparison to Existing Approaches. We finally summarize our work in relation to existing solutions to Spark parameter tuning in terms of the coverage of the mixed parameter space, adaptive runtime optimization, multi-granularity tuning, and multi-objective optimization (MOO), as shown in Table 2. More specifically, ReLocag [16] focuses on individual parameters such as the number of cores but does not cover the broad set of Spark parameters. Search-based solutions to parameter tuning [72, 73] and recent Spark tuning systems [27, 28, 47, 62] cover mixed parameter space but do not support adaptive runtime optimization or multi-granularity tuning. The only system that supports MOO is UDAO [47] but its parameter space is much smaller due to query-level coarse-grained tuning. To the best of our knowledge, our work is the first comprehensive solution to Spark parameter tuning, covering the mixed parameter space with multi-granularity tuning by leveraging the Spark AQE mechanism, and best exploring the tradeoffs between objectives in the multi-objective optimization approach.

4 MODELING

In this section, we introduce our modeling methods that support both compile-time optimization and runtime optimization with fine-grained parameter tuning.

4.1 Compile-time and Runtime Models

We first devise models for fine-grained parameter tuning at both compile-time and runtime.

Runtime models. The Spark optimizer offers the collapsed logical query plan ($\overline{\text{LQP}}$) and query stages (QS) in the physical plan at runtime (see Section 3.1). Hence, we collect these data structures and build runtime $\overline{\text{LQP}}$ and QS models to enable fine-grained tuning of query plan (θ_p) and query stage (θ_s) parameters, respectively.

Compile-time model. At compile time, Spark provides a logical query plan (LQP) and then a physical plan (PQP), but no other

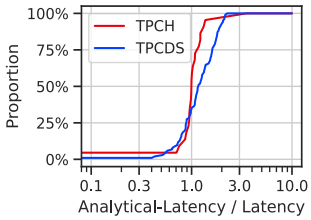


Figure 5: CDF of analytical latency over actual latency

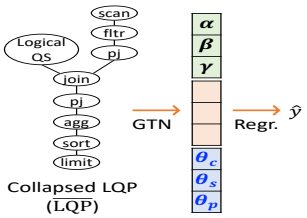


Figure 6: Model structure (GTN embedder + regressor) for $\overline{\text{LQP}}$

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 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 θ_p at compile-time.

4.2 Modeling Objectives

With the objective of optimizing latency and cost, our modeling work seeks to make these metrics more robust and predictable.

Query latency in Spark, defined as the end-to-end duration to execute a query, benefits from Spark's cluster manager that ensures a dedicated allocation of cores and memory to the entire query. Such resource isolation enhances the predictability of latency, making it a suitable target for optimizing a query or a collapsed query plan.

However, within a query, Spark shares resources among parallel query stages, raising two issues in modeling latency at the stage level. First, the end-to-end latency of a set of parallel stages often leads to a longer latency than their maximum due to resource contention. Prior work [32] assumed ample resources in industry-scale clusters and simplified this issue by taking the `max` latency among parallel tasks, which is not applicable in the Spark environment of shared resources. Second, predicting the latency of each stage directly is very hard due to its variability in a shared-resource setting, where performance fluctuates based on resource contention.

To address these issues, we propose the concept of *analytical latency*, calculated as the sum of the task latencies across all data partitions divided by the total number of cores. This yields two advantages. First, it directly links the query latency to its constituent stages, enabling the computation of query-level latency at compile time through a `sum` aggregator over task latencies of all subQs. Second, it enhances the predictability of query stage latency by excluding the variability introduced by resource wait times, thus offering a more consistent basis for latency prediction. To validate the efficacy of the analytical latency, we compare it with actual query latencies using TPC-H and TPC-DS benchmarks under the default Spark configuration. The results demonstrate a robust correlation between analytical and actual latencies, with Pearson correlation coefficients of 97.2% for TPC-H and 87.6% for TPC-DS. As shown in Figure 5, analytical latency closely mirrors actual execution time, with the ratios close to 1 for most of the tested queries.

Cloud costs are mainly based on the consumption of resources, such as CPU-hour, memory-hour, IO and shuffle sizes [4]. We model all of these costs to support multi-objective optimization.

To summarize, our models capture 1) end-to-end latency and cost for collapsed query plans ($\overline{\text{LQP}}$) at runtime, 2) analytical latency and cost for query stages (QS) at runtime, in the face of resource sharing, and 3) analytical latency and cost for subQs at compile time—the latter two ensure both to be robust targets for modeling.

4.3 Model Formulation for Optimization

We now introduce the methodology for building models for subQ, $\overline{\text{LQP}}$, and *QS*, which will enable their respective fine-graining later.

Feature Extraction. We extract features to capture query characteristics and the dynamics of their execution environment, configured by both decision and non-decision variables. First, we extract the query plan as a DAG of vectors, where each query operator is encoded by concatenating i) operator type via one-hot encoding, ii) its cardinality, represented by row count and size in bytes, and iii) the average of the word embeddings [40] from its predicates, providing a rich, multidimensional representation of the operator's functional and data characteristics. Second, we capture critical contextual factors as non-decision variables, including i) input characteristics α, aggregated from the statistics of leaf operators, ii) data distribution β, quantifying the size distribution of input partitions with metrics like standard deviation-to-average ratio ($\frac{\sigma}{\mu}$), skewness ratio ($\frac{\max-\mu}{\mu}$), and range-to-average ratio ($\frac{\max-\min}{\mu}$), and iii) runtime contention γ, capturing the statistics of parallel stages in a numeric vector, tracking their tasks in running and waiting states, and aggregating statistics of completed tasks to characterize their behaviors. Third, we convert the tunable parameters as decision variables into a numeric vector to represent the Spark behavior.

Model Structures. The hybrid data structure of the query plan, with a DAG of operator encodings, and other tabular features, poses a challenge in model formulation. To tackle this, we adopt a multi-channel input framework [32] that incorporates a Graph Transformer Network (GTN) [8] and a regressor to predict our objectives, as shown in Figure 6. We first derive the query embedding using the GTN [8], which handles non-linear and non-sequential relationships by using Laplacian positional encoding to encode positional information and attention mechanisms to capture operator correlations. These embeddings are then concatenated with other tabular data and processed through a regressor, capturing the interplay among the query characteristics, critical contextual factors, and tunable parameters. Figure 6 illustrates the architecture of the $\overline{\text{LQP}}$ model, which has the largest number of feature factors.

5 COMPILE-TIME/RUNTIME OPTIMIZATION

In this section, we present our hybrid compile-time/runtime optimization approach to multi-granularity parameter tuning in the multi-objective optimization setting.

5.1 Hierarchical MOO with Constraints

Our compile-time optimization finds the optimal configuration $\boldsymbol{\theta}_c^*$ of the context parameters to construct an ideal Spark context for query execution. We do so by exploring the correlation of $\boldsymbol{\theta}_c$ with fine-grained $\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}_s$ parameters for different subqueries (subQs), under the modeling constraint that the cardinality estimates are based on Spark's cost-based optimizer. Despite the modeling constraint, capturing the correlation between the mixed parameter space allows us to find a better Spark context for query execution.

The multi-objective optimization problem in Def. 3.3 provides fine-grained control of $\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}_s$ at the subQ/query stage level, besides the query level control of $\boldsymbol{\theta}_c$. This leads to a large parameter space, linear in the number of query stages in the plan, which defeats most existing MOO methods when the solving time must be kept under the constraint of 1-2 seconds for cloud use.

To combat the high-dimensional parameter space, we propose a new approach named *Hierarchical MOO with Constraints* (HMOOC). 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 ($\boldsymbol{\theta}_c$, $\boldsymbol{\theta}_p$, $\boldsymbol{\theta}_s$), one for 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 5.1. Hierarchical MOO with Constraints (HMOOC)

$$\arg\min_{\boldsymbol{\theta}} \quad \boldsymbol{f}(\boldsymbol{\theta}) = \begin{bmatrix} f_1(\boldsymbol{\theta}) = \Lambda(\phi_1(\text{LQP}_1, \boldsymbol{\theta_c}, \boldsymbol{\theta_{p1}}, \boldsymbol{\theta_{s1}}), \ldots, \\ \qquad \phi_1(\text{LQP}_m, \boldsymbol{\theta_c}, \boldsymbol{\theta_{pm}}, \boldsymbol{\theta_{sm}})) \\ \vdots \\ f_k(\boldsymbol{\theta}) = \Lambda(\phi_k(LQP_1, \boldsymbol{\theta_c}, \boldsymbol{\theta_{p1}}, \boldsymbol{\theta_{s1}}), \ldots, \\ \qquad \phi_k(LQP_m, \boldsymbol{\theta_c}, \boldsymbol{\theta_{pm}}, \boldsymbol{\theta_{sm}})) \end{bmatrix}$$

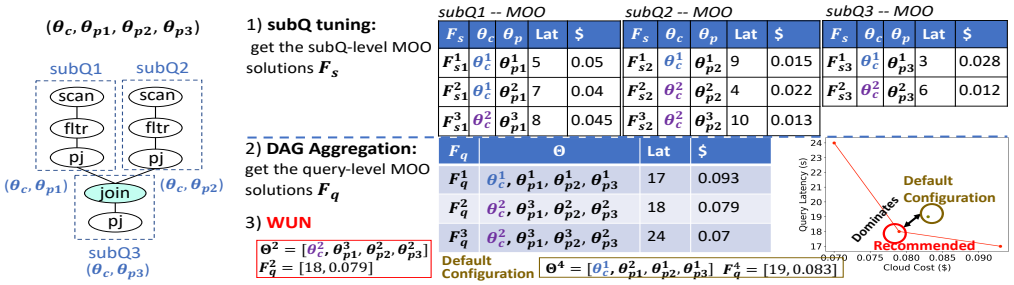
$$s.t. \qquad \boldsymbol{\theta_c} \in \Sigma_c \subseteq \mathbb{R}^{d_c}, \quad \boldsymbol{\theta_{pi}} \in \Sigma_p \subseteq \mathbb{R}^{d_p},$$
$$\boldsymbol{\theta_{si}} \in \Sigma_s \subseteq \mathbb{R}^{d_s}, \quad i = 1, \ldots, m$$

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 $\boldsymbol{\theta_{si}}$. 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.

Our main idea 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 7 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 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 to recommend a configuration from the Pareto front.

5.1.1 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



(a) Example subQ (b) Approach Overview

Figure 7: Example of the Compile-time optimization of TPCH Q3

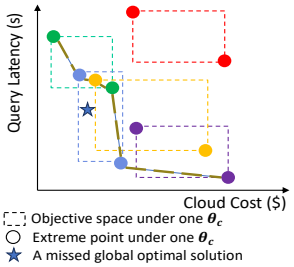


Figure 8: Boundary approximation of DAG optimization

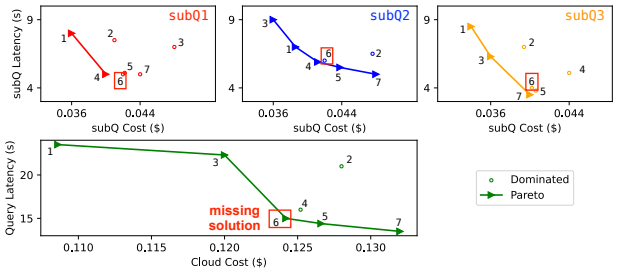


Figure 9: Example of missed global optimal solutions in TPCH Q3

subQ while obeying the constraint that all the subQs share the same θ_c. For simplicity, we focus on (θ_c, θ_p) in the following discussion as θ_s is treated the same way as θ_p.

One may wonder whether it is sufficient to generate only the local Pareto solutions of (θ_c, θ_p) of each subQ. Unfortunately, this will lead to missed global Pareto optimal solutions due to the constraint on θ_c. Figure 9 illustrates an example with 3 subQs, where solutions sharing the same index fall under the same θ_c configuration and have achieved optimal θ_p under that θ_c value. The first row in Figure 9 showcases subQ-level solutions, where triangle points represent subQ-level optima and circle points denote dominated solutions. The second row in Figure 9 displays the global query-level values, where both 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 θ_c constraint and the sum from subQ-level values to query-level values, the sum of solution 6's subQ-level latency and cost becomes better than solution 4 (a subQ-level Pareto optimal solution) and is a query-level Pareto point.

1. Effective θ_c Candidates. To minimize the chance of missing global solutions, we seek to construct a diverse, effective set of θ_c configurations to be considered across all subQs. θ_c can be initialized by random sampling or grid-search over its domain of values. Then, we enrich the θ_c set using a few methods. Drawing inspiration from the evolutionary algorithms [9], we introduce a *crossover* operation over the existing θ_c population to generate new candidates. If crossover cannot generate more candidates, e.g., for some grid search methods used for initial sampling of θ_c candidates, then we add random sampling to discover new candidates.

2. Optimal θ_p Approximation. Next, under each θ_c candidate, we show that it is crucial to keep track of the local Pareto optimal θ_p within each subQ. The following proposition explains why.

Proposition 5.1. Under any specific value θ_c^j, only subQ-level Pareto optimal solutions (θ_c^j, θ_p^*) contribute to the query-level Pareto optimal solutions.

In the interest of the space, all the proofs in this paper are deferred to [31].

The above result allows us to restrict our search of θ_p to only the local Pareto optimal ones. However, given the large, diverse set of θ_c candidates, it is computationally expensive to solve the MOO problem for θ_p repeatedly, once for each θ_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 θ_c candidates entail similar optimal θ_p values in the tuning process. By clustering similar θ_c values into a small number of groups (based on their Euclidean distance), we then solve the MOO problem of θ_p for a single θ_c representative of each group. We then use the optimized θ_p as the estimated optimal solution for other θ_c candidates within each group.

Algorithm. Algorithm 1 describes the steps for obtaining an effective solution set of (θ_c, θ_p) for each subQ. Line 1 initiates by generating $C \times P$ samples, where C and P are the numbers of distinct values of θ_c and θ_p, respectively. The θ_c candidates are then grouped using a clustering approach (Line 2), where *rep_c_list* constitutes the list of θ_c representatives for the n groups, *C_list* includes the members within all n groups, and κ represents the clustering model. In Line 3, θ_p optimization is performed for each representative θ_c candidate (using the samples from Line 1). Subsequently, the optimal θ_p of the representative θ_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 θ_c, and $\Theta^{(0)}$ represents the corresponding configurations. Line 5 further enriches θ_c using the crossover method or random sampling, which expands the initial effective set to generate new θ_c candidates. Afterwards, the cluster model κ assigns the new θ_c candidates with their group labels (Line 6). The previous optimal θ_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) [37]

Algorithm 1: Effective Set Generation

Require: $Q, \phi_i, n, \forall i \in [1,k], \alpha, \beta, \gamma, C, P$.
1: $\Theta_c^{(0)}, \Theta_p^{(0)}$ = sampling(C, P)
2: $rep\_c\_list, C\_list, \kappa$ = cluster($\Theta_c^{(0)}, n$)
3: Θ_p^* = optimize_p_moo($\Theta_p^{(0)}, rep\_c\_list, \phi, \alpha, \beta, \gamma, Q$)
4: $\Omega^{(0)}, \Theta^{(0)}$ = assign_opt_p($C\_list, rep\_c\_list, \Theta_p^*, \phi, \alpha, \beta, \gamma, Q$)
5: Θ_c' = enrich_c($\Omega^{(0)}, \Theta^{(0)}$)
6: C_list' = assign_cluster($\Theta_c', rep\_c\_list, \kappa$)
7: Ω', Θ' = assign_opt_p($C\_list', rep\_c\_list, \Theta_p^*, \phi, \alpha, \beta, \gamma, Q$)
8: Ω, Θ = union($\Omega^{(0)}, \Theta^{(0)}, \Omega', \Theta'$)
9: **return** Ω, Θ

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 [31] for more details.

5.1.2 DAG Aggregation. DAG aggregation aims to recover query-level Pareto optimal solutions from subQ-level solutions. It is a combinatorial MOO problem, as each subQ must select a solution from its non-dominated solution set while satisfying the constraint of sharing the θ_c configuration among all subQs. The complexity of this process can be exponential in the number of subQs. Our 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 our 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. The MOO problem over a DAG can be simulated with a list structure for computing query-level objectives.

HMOOC1: Divide-and-Conquer. Under a fixed θ_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 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. Our proof (available in [31]) shows that this method returns a full set of query-level Pareto optimal solutions.

HMOOC2: WS-based Approximation. Our second technique approximates the MOO solution over a list structure. For each fixed θ_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 Pareto solutions (see [31]).

HMOOC3: Boundary-based Approximation. Our next approximate technique stems from the idea that the objective space of DAG aggregation under each θ_c can be approximated by k *extreme points*, where k is the number of objectives. In our context, the *extreme point* under a fixed θ_c is the Pareto optimal point with the best query-level value for any objective. The rationale behind this approximation lies in the observation that solutions from different θ_c candidates correspond to distinct regions on the query-level Pareto front. This arises from the fact that each θ_c candidate determines the total resources allocated to the query, and a diverse set of θ_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 boundaries of different (resource) regions within the query-level Pareto front. Figure 8 illustrates an example. Here, the dashed rectangles with their extreme points under different colors represent the objective space of query-level solutions under various θ_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 θ_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 query-level *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 θ_c candidates. An additional filtering step will retain the non-dominated solutions from the kn candidates, using an existing method of complexity $O(kn\log(kn))$ [22].

Proposition 5.2. Under a fixed θ_c candidate, the query-level objective space of Pareto optimal solutions is bounded by its extreme points in a 2D objective space.

Proposition 5.3. 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.

5.1.3 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 θ_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.

5.2 Runtime Optimization

The compile-time optimization relied on the estimated cardinality and assumption of uniform data distributions. However, its true value is to recommend the optimal context parameters θ_c^* by considering the correlations with θ_p and θ_s. Then, our runtime optimization addresses the remaining problems, adapting θ_p and θ_s based on actual runtime statistics and plan structures.

To start the runtime process, we need to first suit the constraint that Spark accepts only one copy of θ_p and θ_s at the query submission time. To do so, we intelligently aggregate the fine-grained θ_p and θ_s from compile-time optimization to initialize the runtime process. In particular, Spark AQE can convert a sort-merge join (SMJ) to a shuffled hash join (SHJ) or a broadcast hash join (BHJ), but not vice versa. Thus, imposing high thresholds (s_3, s_4 in Table 1) to force SHJ or BHJ based on inaccurate compile-time cardinality can result in suboptimal plans. In the example of Figure 3(b), when the cardinality of `Join4` is underestimated at the compile time, MO-WS returns a query plan broadcasting the output of `Join4`. At runtime, when the actual output size of `Join4` is observed to be 4.5GB, Spark cannot switch the BHJ back to other joins but broadcasting the 4.5GB data, leading to suboptimal performance. On the other hand, setting s_3, s_4 to zeros initially might overlook opportunities to apply BHJs, especially for joins on base tables with small input sizes. To mitigate this, we initialize θ_p with the smallest threshold among all join-based subQs, enabling more effective runtime decisions. Other details of aggregating θ_p and θ_s are in [31].

Runtime optimization then operates within a client-server model. The client, integrated with the Spark driver, dispatches optimization requests—including runtime statistics and plan structures—when a collapsed logical query plan ($\overline{\text{LQP}}$) or a runtime query stage (QS) necessitates optimization (Steps 6, 9 in Figure 2). The server, hosted on a GPU-enabled node and supported by the learned models and a library of MOO algorithms [47], processes these requests over a high-speed network connection.

Complex queries can trigger numerous optimization requests every time when a collapsed logical plan or a runtime QS is produced, significantly impacting overall latency. For instance, TPC-DS queries, with up to 47 subQs, may generate up to nearly a hundred requests throughout a query's lifecycle. To address this, we established rules to prune unnecessary requests based on the runtime semantics of parametric rules as detailed in [31]. By applying these rules, we substantially reduce the total number of optimization calls by 86% and 92% for TPC-H and TPC-DS respectively.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate our modeling and fine-grained compile-time/runtime optimization techniques. We further present an end-to-end evaluation against the SOTA tuning methods.

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 θ_c, 9 for θ_p, and 2 for θ_s, selected based on feature selection profiling [19] and best practices from the Spark documentation. More details are in [31].

Table 3: Model performance with Graph Transofmer Network (GTN) + Regressor

	Target	Ana-Latency/Latency (s) WMAPE	P50	P90	Corr	Shuffle Size (MB) WMAPE	P50	P90	Corr	Xput K/s
TPC-H	subQ	0.131	0.029	0.292	0.99	0.025	0.006	0.045	1.00	70
	QS	0.149	0.027	0.353	0.98	0.002	3e-05	0.004	1.00	86
	$\overline{\text{LQP}}$	0.164	0.060	0.337	0.95	0.010	8e-05	0.002	1.00	146
TPC-DS	subQ	0.249	0.030	0.616	0.95	0.098	0.016	0.134	0.99	60
	QS	0.279	0.060	0.651	0.95	0.028	4e-04	0.023	1.00	79
	$\overline{\text{LQP}}$	0.223	0.095	0.459	0.93	0.107	0.028	0.199	0.99	462
TPC-H* (TPC-DS)	subQ	0.139	0.018	0.346	0.984	0.019	0.002	0.034	0.996	45
	QS	0.143	0.017	0.346	0.982	0.003	0.001	0.006	1.000	72
	$\overline{\text{LQP}}$	0.148	0.046	0.292	0.924	0.019	0.005	0.028	0.998	242

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 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 [37].

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 per query. (2) The compile-time optimizer operates as a standalone module, costing an average of 0.4s. (3) The runtime optimizer operates in the server-client model, where the client is wrapped in customized Spark query plan rules, and the server is on a standalone machine, costing an average of 0.3-0.4s per query. (4) 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.

6.1 Model Evaluation

We trained separate models for subQ, QS, and $\overline{\text{LQP}}$ to support compile-time/runtime optimization. We evaluate each model using the weighted mean absolute percentage error (WMAPE), median and 90th percentile errors (P50 and P90), Pearson correlation (Corr), and inference throughput (Xput).

Expt 1: *Model Performance.* We present the performance of our best-tuned models for TPC-H and TPC-DS in the first 6 rows of Table 3. First, our models can provide highly accurate prediction in latency and analytical latency for Spark queries for different compile-time and runtime targets, achieving WMAPEs of 13-28%, P50 of 3-10%, and P90 of 29-65%, alongside a correlation range of 93-99% with the ground truth. Second, the shuffle size is more predictable than latency, evidenced by a WMAPE of 0.2-11% and almost perfect 99-100% correlation with the actual shuffle size, attributed to its consistent performance across configurations. Third, the models show high inference throughput, ranging from 60-462K queries per second, which enables efficient solving time of our compile-time and runtime optimizations.

Expt 2: *Impact of Skewness.* We now investigate the impact of data skewness on our latency models. Following recent work [21], we define the skewness ratio as the gap between the maximum and

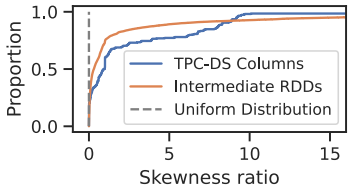


Figure 10: CDF of TPC-DS skewness

Table 4: Correlation between model errors and skewness ratios in intermediate RDDs.

Lat. range	>1s	$>Lp_{90}$	$>Lp_{99}$
TPC-H	0.001	0.019	-0.026
TPC-DS	-0.017	-0.015	0.048

Table 5: Latency reduction with a strong speed preference

	TPC-H			TPC-DS		
	MO-WS	HMOOC3	HMOOC3+	MO-WS	HMOOC3	HMOOC3+
Total Lat Reduction	18%	61%	63%	25%	61%	65%
Avg Lat Reduction	-1%	53%	52%	34%	56%	59%
Avg Solving Time (s)	2.6	0.41	0.70	15	0.41	0.80
P50 Solving Time (s)	2.3	0.38	0.69	14	0.35	0.65
P90 Solving Time (s)	4.0	0.51	0.90	26	0.63	1.4
Max Solving Time (s)	4.5	0.81	1.4	68	1.4	2.9

average partition size divided by the average. Figure 10 shows the skewness ratio in base tables and intermediate RDDs in the TPC-DS benchmark, which exceeds 100% for over 40% of columns in base tables and for over 26% of the intermediate RDDs. Table 4 further shows that the correlations between model errors and skewness are close to 0 during runtime optimization, across queries in different latency ranges. This indicates that once the skewness is captured at runtime, the model can work quite well regardless of the skewness level. Additional results are reported in [31].

Expt 3: *Model Generalizability.* We further explore the generalizability of our trained model to unseen workloads. The first approach applies the latency model trained on TPC-DS directly to TPC-H, or vice versa. However, it leads to high errors due to significant differences in the execution environments. The second approach transfers the graph embedding through the GTN model and retrains only the regressor for latency prediction for a different workload. Our results indicate that graph embeddings trained on a workload with a broader range of query operators (e.g., TPC-DS) can be transferred effectively. As shown in the last three rows of Table 3, applying the GTN learned from TPC-DS to TPC-H queries and retraining the regressor for TPC-H leads to a modest 0.008 WMAPE increase for the subQ model and a 0.006 and 0.016 WMAPE decrease for QS and LQP, respectively. More details can be found in [31].

6.2 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 4: *DAG Aggregation methods.* We first compare the three DAG aggregation methods (§5.1.2) 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 and hence we omit the plot in the interest of space. For efficiency, Figure 11(a) 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.

Expt 5: *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 11(b)-11(c) 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, `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.

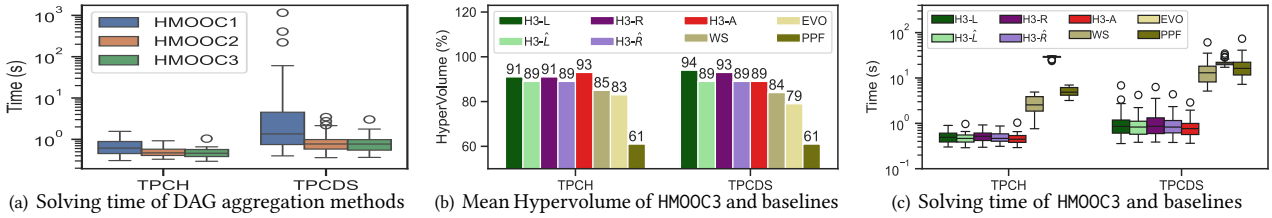
Expt 6: *Comparison with SOTA MOO methods.* We next compare with SOTA MOO methods, WS [35] (with tuned hyperparameters of 10k samples and 11 pairs of weights), Evo [9] (with a population size of 100 and 500 function evaluations), and PF [47], for fine-grained tuning of parameters based on Def. 3.3. Figures 11(b)-11(c) 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.

6.3 End-to-End Evaluation

We now extend our best compile-time optimization method `HMOOC3` with runtime optimization, denoted as `HMOOC3+`, which includes the adaptive sampling rate scheme introduced in Section 5.1 to achieve a better balance between accuracy and efficiency. We compare it with existing methods in actual execution time when Spark AQE is enabled. To account for model errors, we refine the search range for each Spark parameter by avoiding the extreme values of the parameter space that could make the predictions less reliable.

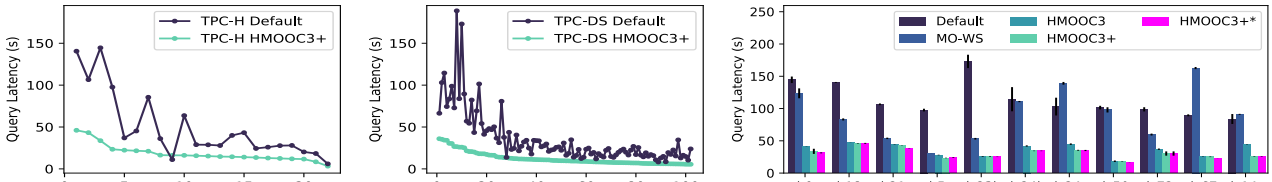
Expt 7: *Benefits over Default Configuration.* Figure 12(a) and 12(b) show the per-query latency using `HMOOC3+` or the default configuration. Among the 22 TPC-H and 102 TPC-DS queries, `HMOOC3+` significantly outperforms the default configuration for most of the queries, while being similar (within 2s latency improvement) for three short-running queries (`h6`, `ds8`, and `ds20`). `HMOOC3+` loses to the default configuration only for one short-running query (`h14`), mainly due to the model error in predicting its latency. As we use WMAPE as the training loss to give more weight to long-running queries, short queries can have inaccurate predictions and hence experience a suboptimal query plan or insufficient resource allocation.

Expt 8: *Benefits over Query-level MOO.* We next show the advantages of our methods (`HMOOC3` and `HMOOC3+`) over the best-performing MOO method identified in the previous study, i.e., WS for query-level MOO, denoted as `MO-WS`. Here, we prioritize latency over cost with a preference vector of $(0.9, 0.1)$ on latency and cost. The results in Table 5 show the improvement over the default Spark configuration. First, *fine-grained tuning significantly enhances performance* (major result **R1**), cutting latency by 61% for



(a) Solving time of DAG aggregation methods (b) Mean Hypervolume of HMOOC3 and baselines (c) Solving time of HMOOC3 and baselines

Figure 11: Accuracy and efficiency of our compile-time MOO algorithms, compared to existing MOO methods



(a) Latency of queries in TPC-H (b) Latency of queries in TPC-DS (c) Long-running queries in TPC-H and DS

Figure 12: End-to-end performance of our algorithm, compared to the state-of-the-art (SOTA) methods

Table 6: Latency and cost adapting to preferences

Prefs. Lat/Cost	TPC-H SO-FW	TPC-H HMOOC3+*	TPC-DS SO-FW	TPC-DS HMOOC3+*
(0.0, 1.0)	20% / -11%	-15% / -10%	-6% / 64%	-45% / -22%
(0.1, 0.9)	1% / 1%	-31% / -5%	-28% / 105%	-57% / -7%
(0.5, 0.5)	-1% / 25%	-46% / -3%	-28% / 128%	-59% / 29%
(0.9, 0.1)	-13% / 27%	-51% / 0%	-34% / 139%	-59% / 55%
(1.0, 0.0)	-14% / 44%	-55% / 3%	-26% / 144%	-59% / 64%

both benchmarks with compile-time optimization (HMOOC3), and by 63-65% with runtime optimization (HMOOC3+). They both outperform MO-WS with only 18-25% reductions and in some cases, worse than the default configuration. Second, MO-WS, *even limited to query-level tuning, suffers in efficiency*, with 14s and 26s as the P50 and P90 solving time, respectively, for TPC-DS. In contrast, our approach solves MOO with a P50 solving time of 0.65-0.69s and P90 time of 0.9-1.4s for the two benchmarks.

Expt 9: *Benefits for Long-running Queries.* We further consider long-running queries that are hard to optimize and often suffer from suboptimal plans with θ_p tuned based on compile-time cardinality estimates. Figure 12(c) shows the top long-running queries from TPC-H and TPC-DS. While HMOOC3 already offers significant latency reduction, HMOOC3+ achieves up to a 22% additional latency reduction over the default configuration. We finally turned on multi-query plan search, running up to 3 query plan structures in parallel at compile-time optimization. This method, HMOOC3+*, further improves h9, h21, ds50, etc. Overall, HMOOC3+* *reduces latency by 64-85% for long-running queries* (**R2**).

Expt 10: *Adaptability Comparison to SO with fixed weights.* As MO-WS is too slow for cloud use due to its inefficiency, we now compare the adaptability of our approach against the common, practical approach that combines multiple objectives into a single objective using fixed weights [25, 66, 73], denoted as SO-FW. Table 6 shows the average reduction rates in latency and cost relative to the default configurations across a range of preference vectors. First, HMOOC3+* *dominates* SO-FW *with more latency and cost reductions in most cases* (**R3**), achieving up to 55-59% latency reduction and 10-22% cost reduction in both benchmarks, while SO-FW gets at most 1-34% average latency reduction and in most cases, increases the cost compared to default. Second, *our approach demonstrates superior adaptability to varying preferences* (**R4**), enhancing latency reductions progressively as preferences shift towards speed. In contrast, SO-FW does not make meaningful recommendations: Under a cost-saving preference of (0.0, 1.0), SO-FW struggles to lower costs in TPC-DS, instead increasing the average cost by 64% across all queries. Despite this cost increase, it achieves a merely 6% reduction in latency. In contrast, HMOOC3+* achieves a 45% reduction in latency alongside a 22% cost saving, underscoring its effectiveness and adaptability to the specified cost performance preference.

7 CONCLUSIONS

This paper presented a Spark optimizer for fine-grained parameter tuning in the new AQE architecture based on a hybrid compile-time/runtime optimization approach. Our approach employed sophisticated modeling techniques to capture different compile-time and runtime modeling targets, and a suite of techniques tailored for multi-objective optimization (MOO) while meeting the stringent solving time constraint of 1-2 seconds. Evaluation results using TPC-H and TPC-DS benchmarks show that (*i*) when prioritizing latency, our approach achieves 63% and 65% latency reduction on average for TPC-H and TPC-DS, respectively, under the solving time of 0.7-0.8 sec, outperforming the most competitive MOO method with 18-25% latency reduction and high solving time of 2.6-15 sec; (*ii*) when shifting preferences between latency and cost, our approach dominates the solutions from alternative methods by a wide margin. In the future, we plan to extend our tuning approach to support diverse (e.g., machine learning) workloads. In addition, we plan to extend our approach to other big data/DMBS systems like Presto [49], Greenplum [33], and MaxCompute [36], which can observe runtime statistics and support runtime adaptability.

ACKNOWLEDGMENTS

This work was partially supported by the European Research Council (ERC) Horizon 2020 research and innovation programme (grant n725561) and China Scholarship Council (CSC). We also thank Julien Fontanarava for engineering efforts and Guillaume Lachaud for the discussion and help.

REFERENCES

[1] Kavita Agarwal, Bhushan Jain, and Donald E. Porter. 2015. Containing the Hype. In *Proceedings of the 6th Asia-Pacific Workshop on Systems, APSys 2015, Tokyo, Japan, July 27-28, 2015*, Kenji Kono and Takahiro Shinagawa (Eds.). ACM, 8:1–8:9. https://doi.org/10.1145/2797022.2797029

[2] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. 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*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1383–1394. https://doi.org/10.1145/2723372.2742797

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

[4] Google Cloud. 2022. Dataflow Pricing. https://cloud.google.com/dataflow/pricing

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

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

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

[8] Vijay Prakash Dwivedi and Xavier Bresson. 2021. A Generalization of Transformer Networks to Graphs. *AAAI Workshop on Deep Learning on Graphs: Methods and Applications* (2021).

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

[10] Wenchen Fan, Herman van Hovell, and MaryAnn Xue. 2020. 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.

[11] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. 2020. 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* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 2494âĂŞ2504. https://doi.org/10.1145/3394486.3403299

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

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

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

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

[16] Zhiyao Hu, Dongsheng Li, Dongxiang Zhang, Yiming Zhang, and Baoyun Peng. 2021. Optimizing Resource Allocation for Data-Parallel Jobs Via GCN-Based Prediction. *IEEE Trans. Parallel Distributed Syst.* 32, 9 (2021), 2188–2201. https://doi.org/10.1109/TPDS.2021.3055019

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

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

[19] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. 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*, Anirudh Badam and Vijay Chidambaram (Eds.). USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/kanellis

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

[21] Paraschos Koutris and Dan Suciu. 2016. A Guide to Formal Analysis of Join Processing in Massively Parallel Systems. *SIGMOD Rec.* 45, 4 (2016), 18–27. https://doi.org/10.1145/3092931.3092934

[22] Hsiang-Tsung Kung, Fabrizio Luccio, and Franco P Preparata. 1975. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)* 22, 4 (1975), 469–476.

[23] Mayuresh Kunjir and Shivnath Babu. 2020. 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*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1667–1683. https://doi.org/10.1145/3318464.3380591

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

[25] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[26] Rundong Li, Ningfang Mi, Mirek Riedewald, Yizhou Sun, and Yi Yao. 2019. Abstract cost models for distributed data-intensive computations. *Distributed Parallel Databases* 37, 3 (2019), 411–439. https://doi.org/10.1007/S10619-018-7244-2

[27] Yang Li, Huaijun Jiang, Yu Shen, Yide Fang, Xiaofeng Yang, Danqing Huang, Xinyi Zhang, Wentao Zhang, Ce Zhang, Peng Chen, and Bin Cui. 2023. Towards General and Efficient Online Tuning for Spark. *Proc. VLDB Endow.* 16, 12 (2023), 3570–3583. https://doi.org/10.14778/3611540.3611548

[28] Chen Lin, Junqing Zhuang, Jiadong Feng, Hui Li, Xuanhe Zhou, and Guoliang Li. 2022. 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, 1995–2007. https://doi.org/10.1109/ICDE53745.2022.00195

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

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

[31] Chenghao Lyu, Qi Fan, Philippe Guyard, and Yanlei Diao. 2024. *A Spark Optimizer for Adaptive, Fine-Grained Parameter Tuning*. Technical Report. https://chenghao.pages.dev/papers/vldb24-lyu-tr.pdf

[32] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, and Jingren Zhou. 2022. Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing. *Proc. VLDB Endow.* 15, 11 (2022), 3098–3111. https://doi.org/10.14778/3551793.3551855

[33] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. 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*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2530–2542. https://doi.org/10.1145/3448016.3457562

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

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

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

[37] Michael D. McKay, Richard J. Beckman, and William J. Conover. 2000. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output From a Computer Code. *Technometrics* 42, 1 (2000), 55–61.

https://doi.org/10.1080/00401706.2000.10485979

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

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

[40] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (Eds.). 3111–3119. https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html

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

[42] Vikram Nathan, Vikramank Singh, Zhengchun Liu, Mohammad Rahman, Andreas Kipf, Dominik Horn, Davide Pagano, Balakrishnan Narayanaswamy Gaurav Saxena, and Tim Kraska. [n.d.]. Intelligent Scaling in Amazon Redshift. In *SIGMOD '24: International Conference on Management of Data, Philadelphia, 2024*. ACM, 1–. To appear.

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

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

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

[46] Prateek Sharma, Lucas Chaufournier, Prashant J. Shenoy, and Y. C. Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*. ACM, 1. http://dl.acm.org/citation.cfm?id=2988337

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

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

[49] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, Nikhil Collooru, Ke Wang, Sameer Agarwal, Arjun Gupta, Dionysios Logothetis, Kostas Xirogiannopoulos, Amit Dutta, Varun Gajjala, Rohit Jain, Ajay Palakuzhy, Prithvi Pandian, Sergey Pershin, Abhisek Saikia, Pranjal Shankhdhar, Neerad Somanchi, Swapnil Tailor, Jialiang Tan, Sreeni Viswanadha, Zac Wen, Biswapesh Chattopadhyay, Bin Fan, Deepak Majeti, and Aditi Pandit. 2023. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM Manag. Data* 1, 2 (2023), 189:1–189:25. https://doi.org/10.1145/3589769

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

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

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

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

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

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

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

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

[58] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization using Reinforcement Learning. *Proc. VLDB Endow.* 14, 13 (2021), 3402–3414. https://doi.org/10.14778/3484224.3484236

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

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

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

[62] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD/PODS âĂŹ22)*. ACM. https://doi.org/10.1145/3514221.3526157

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

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

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

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

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

[68] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards Dynamic and Safe Configuration Tuning for Cloud Databases. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 631–645. https://doi.org/10.1145/3514221.3526176

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

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

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

[72] Yuqing Zhu and Jianxun Liu. 2019. ClassyTune: A Performance Auto-Tuner for Systems in the Cloud. *IEEE Transactions on Cloud Computing* (2019), 1–1.
[73] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: tapping the performance potential of systems via automatic configuration tuning. *SoCC '17: ACM Symposium on Cloud Computing Santa Clara California September, 2017* (2017), 338–350.