A Spark Optimizer for Adaptive, Fine-Grained Parameter Tuning

Chenghao Lyu[†] Qi Fan[‡] Philippe Guyard[‡] Yanlei Diao[†‡]
[†] University of Massachusetts, Amherst [‡]Ecole Polytechnique
chenghao@cs.umass.edu,{qi.fan,philippe.guyard,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 (collectively called a "configuration") 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 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. Our evaluation results using the TPC-H and TPC-DS benchmarks demonstrate the superior performance of our approach: (*i*) When prioritizing latency, it achieves an average of 61% and 64% reduction for TPC-H and TPC-DS, respectively, under the solving time of 0.62-0.83 sec, outperforming the most competitive MOO method that reduces only 18-25% latency with high solving time of 2.4-15 sec. (*ii*) When shifting preferences between latency and cost, our approach dominates the solutions from alternative methods by a wide margin, exhibiting superior adaptability to varying preferences.

1 INTRODUCTION

Big data query processing has become an integral part of enterprise businesses and many platforms have been developed for this purpose [3, 5, 6, 11, 14, 31, 36, 44, 49, 56, 57, 62, 63]. As these systems are becoming increasingly complex, parameter tuning of big data systems has recently attracted a lot of research attention [21, 23–25, 41, 55]. Take Apache Spark for example. It offers over 180 parameters for governing a *mixed set of decisions*, including resource allocation, the degree of parallelism, IO and shuffling behaviors, and SQL-related decisions based on parametric query optimization rules. 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 before planning the next stage. The recent work [28] has explored this opportunity to optimize the resource allocation of each query stage, but is limited to two (CPU and memory) resource parameters of each parallel instance of a stage. Recently, Spark has taken a step further to introduce Adaptive Query Execution (AQE), which enables the logical query plan to be re-optimized upon the completion of each query stage and the query stages produced from the newly generated physical plan to be re-optimized as well, both based on 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 suffering from suboptimal performance of AQE when the parameters are set to inappropriate values. On the other hand, recent work on Spark tuning [21, 23–25, 41, 55] 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 [28, 37] has become more critical than ever to end users. Prior work [23, 59, 66] 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 [30] pointing out that solving such a SO problem is unlikely to return a solution that balances the cost-performance in the objective space as the specified weights intend to express (as we will demonstrate in this work). The theoretically sound approach is to treat it as a *multi-objective optimization* (MOO) problem [8, 30, 33, 34], compute the Pareto optimal set, and return one solution from the Pareto set that best matches the user preference as reflected by the weights set on the cost-performance objectives [28, 41].

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 theoretically sound multi-objective optimization setting to better adapt to user cost-performance needs.* This Optimizer for Parameter Tuning (OPT) complements Spark's current Cost-based and Rule-based Optimization (CRO) of query plans, where the optimization rules 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 to Spark 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 governing runtime behaviors such as shuffling; 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 daunting challenge is solving the MOO problem over a large parameter space in the complex Spark environment while obeying stringent time constraints for cloud use. In particular, the solving time of MOO must be kept under the time constraint of 1-2 seconds to avoid delaying the launch of a Spark application in cloud execution, as recently emphasized for serverless computing [28]. Prior work on MOO for Spark tuning [41] has reported the running time of the Evolutional (Evo) method [8] to be about 5 seconds for query-level control of the most important 12 Spark parameters. However, when we increase the parameter space to allow the θ_p parameters to be tuned separately for different subqueries, the time cost of Evo goes up quickly, exceeding 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 multi-granularity tuning (Section 3): Our OPT is designed for multi-granularity tuning of a mixed parameter space: 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 the performance gains. To cope with the different control mechanisms that Spark provides 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 among 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 to parameter tuning requires accurate models for both compile-time and runtime optimization, where the query plans have different representations in these two phases. 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 further provide a suite of techniques that derive both 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, which must be set at the query level to enable runtime resource sharing. 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.

We performed an extensive evaluation of our modeling and MOO techniques using the TPC-H and TPC-DS benchmarks. (1) *Modeling*: Our compile-time and runtime models consistently provide accurate predictions for Spark queries, with the weighted mean absolute percentage error between 13-28% in latency and 0.2-10.7% in IO cost. (2) *MOO algorithms*: Our compile-time MOO algorithm (HMOOC) for fine-grained parameter tuning outperforms existing MOO methods with 7.9%-81.7% improvement in hypervolume (the dominated space covered by the Pareto front) and 81.8%-98.3% reduction in solving time. (3) *End-to-end evaluation*: We further add runtime optimization, denoted as HMOOC+, and compare its recommended configuration with those returned by other competitive solutions. When prioritizing latency, HMOOC+ achieves an average of 61% and 64% reduction for TPC-H and TPC-DS, respectively, and an average solving time of 0.62-0.83s, outperforming the most competitive MOO method, which only reduces 18-25% latency with high solving time of 2.4-15s. When shifting preferences between latency and cost, HMOOC+ dominates the only available efficient method, single-objective weighted sum, in both latency and cost reductions, 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 [48, 51, 59, 60], which can take long to run (e.g., 15-45 minutes [48, 59]). Ottertune [48] 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 [60] 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 [59] and QTune [23] use Deep Reinforcement Learning (RL) to predict the reward of a configuration, which is a scalar value composed of different objectives (e.g., latency and

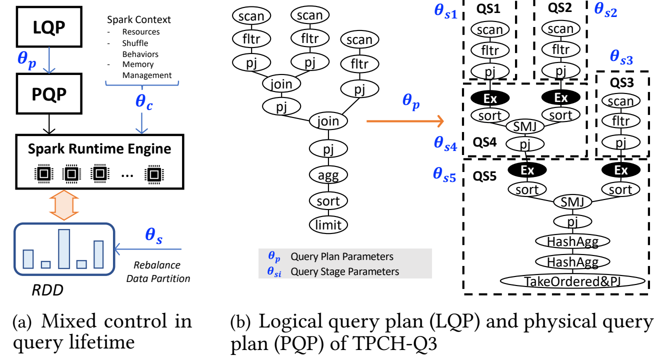
throughput) based on fixed weights, and explores new configurations to optimize the reward. These methods can take many iterations to achieve good performance [55]. UDO [51] is an offline RL-based tuner for both database physical design choices and parameter settings. Unlike the prior work, OnlineTuner [61] 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 a plugin 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. Further, none of the above methods can be applied to adaptive, fine-grained runtime optimization of Spark jobs and are limited to single-objective optimization.

Tuning of big data systems. Among *search-based* methods, BestConfig [66] searches for good configurations by dividing high-dimensional configuration space into subspaces based on samples, but it cold-starts each tuning request. ClassyTune [65] solves the optimization problem by classification, which cannot be easily extended to the MOO setting. A new line of work has considered parameter tuning for Spark, specifically, for recurring workloads that are observed repeatedly under different configurations. ReIM [21] addresses online tuning of memory management decisions by guiding the GP approach using manually-derived memory models. Locat [55] is a data-aware GP-based approach for tuning Spark queries that repeatedly run with the input data size changing over time. While it is shown to outperform prior solutions such as Tuneful [10], ReIM [21], and QTune [23] in efficiency, it still needs hours to complete. Li et al. [24] further tune periodic Spark jobs using a GP with safe regions and meta-learning from the history. LITE[25] tunes parameters of non-SQL Spark applications and relies on stage code analysis to derive predictive models, which is impractical as the cloud providers usually have no access to application code due to 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 [17] 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 [40] optimizes latency via an exhaustive search of the solution space while calling its model for predicting the performance of each solution. WiseDB [29] manages cloud resources based on a decision tree trained on minimum-cost schedules of sample workloads. ReLocag[15] presents a predictor to find the near-optimal number of CPU cores to minimize job completion time. Recent work [22] 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 a 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,



(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

aka, the Pareto-optimal set (or Pareto front). Theoretical MOO solutions suffer from various performance issues in cloud optimization: Weighted Sum [30] is known to have *poor coverage* of the Pareto front [33]. Normalized Constraints [34] lacks in *efficiency* due to repeated recomputation to return more solutions. Evolutionary methods [8] approximately compute a Pareto set but suffer from *inconsistent solutions*. Multi-objective Bayesian Optimization [4, 13] extends the Bayesian approach to modeling an unknown function with an acquisition function for choosing the next point(s) that are likely to be Pareto optimal. But it is shown to take long to run [41] and hence lacks the *efficiency* required by a cloud optimizer.

In the DB literature, MOO for SQL queries [16, 19, 45–47] 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 [43] considers multiple SLOs of SQL queries and guarantees max-min fairness when they cannot be all met. MOO for workflow scheduling [19] 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 [41, 58] that tunes Spark configurations to optimize for multiple objectives. It *Progressive Frontier* (PF) method [41] 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 [28] 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 formally define our Spark parameter tuning problem and provide an overview of our approach.

3.1 Background on Spark

Apache Spark [57] 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,

Table 1: (Selected) Spark parameters in three categories

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

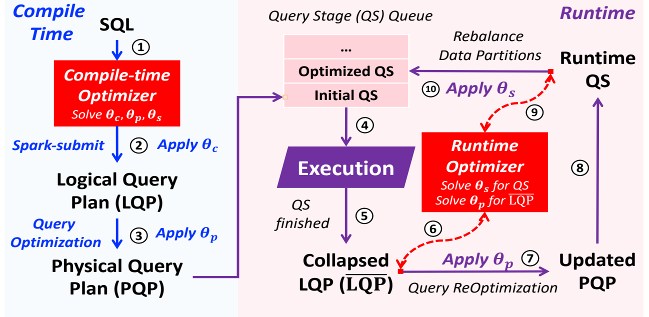


Figure 2: Query life cycle with an optimizer for parameter tuning

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 data processing into the Spark framework [1]. A submitted SQL query undergoes parsing, analysis, and optimization to form a *logical query plan* (LQP). In subsequent physical planning, Spark takes the LQP and generates 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 physical 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 the data exchange dependencies such as shuffling or broadcasting. These query stages are executed in a topological order, manifesting themselves as Spark jobs.

The execution of a Spark SQL query is configured by three categories of parameters, providing mixed control through the query lifetime. Table 1 shows the selected parameters from each category, and Figure 1 illustrates how they are applied in a query lifetime. As Figure 1(a) shows, **query plan parameters** θ_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 through parametric optimization rules in the Spark optimizer. 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 over the cluster, and query stages are identified at the boundaries of exchange operators. Further, **query stage parameters** θ_s control the optimization of a query stage via parametric rules, such as rebalancing data partitions. Finally, **context parameters** θ_c, specified on the Spark context, control shared resources, shuffle behaviors, and memory management through the entire SQL execution. Although they are in effect only during query execution, they must be specified at the query submission time when the Spark context is initialized.

Adaptive Query Execution (AQE). Cardinality estimation [12, 26, 27, 38, 39, 42, 50, 52–54, 64] has been a long-standing issue that impacts the effectiveness of the physical query plan. To address this issue, Spark has recently introduced *Adaptive Query Execution* (AQE) that enables runtime optimization based on precise statistics collected from completed stages at runtime [9]. Figure 2 shows the life cycle of an SQL query with the AQE mechanism turned on. At compile time, a query is transformed to a logical query plan (LQP) and then a physical query plan (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 their observed cardinalities, leading to a so-called collapsed query plan $\overline{\text{LQP}}$ (step 5), and re-optimizes the $\overline{\text{LQP}}$ (step 7) and the QSs (step 10), until all QSs are completed.

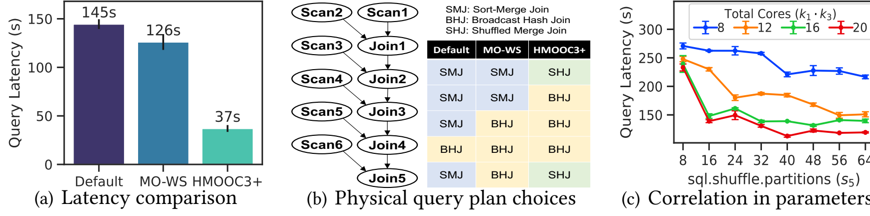
At the core of AQE are runtime optimization rules. Each rule internally traverses the query operators and takes effect on them. These rules are categorized as parametric and non-parametric, and each parametric rule is configured by a subset of θ_p or θ_s parameters. The details of those rules are left to Appendix B.1.2.

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, by adding query-level tuning of parameters using prior MOO work (`MO-WS`) [41] and then running AQE can already provide 13% improvement over AQE using the default configuration.

Second, *fine-grained control has performance benefits over query-level coarse-grained control*. All existing work on Spark parameter tuning [21, 23–25, 41, 55] focuses on query-level control: that is, one copy of θ_c, θ_p and θ_s parameters are applied to all operators in the logical query plan and all the stages in the physical plan. However, we observe that adapting θ_p for different collapsed query plans and θ_s for different query stages offer additional performance benefits, which were missed in all prior solutions. Figure 3(a) further shows that adapting θ_p for different collapsed query plans during runtime can lead to better performance than query-level tuning of θ_p, further reducing the latency by 61%. Figure 3(b) shows the simplified query structure of TPCH-Q9, including 6 scan operators and 5 join operators. By adapting θ_p for different collapsed query plans with observed statistics at runtime, we manage to construct a new physical query plan with 3 broadcast hash joins (BHJs) and 2 shuffled hash joins (SHJs), outperforming the query-level θ_p choice involving 2 sort-merge joins (SMJ) + 3 BHJs. Specifically, `MO-WS`



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

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

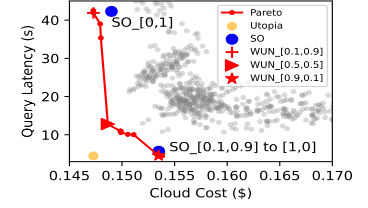


Figure 4: MOO solutions for TPCH Q2

broadcasts up to 4.5G data in Join5 because it finalized its parameter tuning at compile time with underestimated cardinality of Join4, while our fine-grained tuning can derive a better plan by adapting $\boldsymbol{\theta}_p$ to runtime statistics.

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.* One may wonder whether fine-grained parameter tuning can be conducted solely at runtime, as an augmented AQE process. Unfortunately, Spark parameter tuning is more complex than that: the $\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}_s$ parameters are best tuned at runtime to benefit from precise statistics, but 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. For example, 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

In this section, we introduce our parameter tuning approach that is grounded in two principles:

3.3.1 Hybrid, Multi-Granularity Tuning. The goal of this paper is to find the optimal configuration of all the $\boldsymbol{\theta}_p$, $\boldsymbol{\theta}_s$, and $\boldsymbol{\theta}_c$ parameters of each Spark query. A key feature of our approach is *multi-granularity tuning*: While the context parameters $\boldsymbol{\theta}_c$ configure the Spark context at the *query level*, we aim to tune the $\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}_s$ parameters at fine granularity to maximize the performance gains. More precisely, the query plan parameters $\boldsymbol{\theta}_p$ can be tuned for each *collapsed query plan*, and the query stage parameters $\boldsymbol{\theta}_s$ can be tuned for each *query stage* in the physical query plan.

To address the correlation between the context parameters $\boldsymbol{\theta}_c$, which must be set at query submission time, and $\boldsymbol{\theta}_p$ and $\boldsymbol{\theta}_s$ parameters, which are best tuned during AQE with precise statistics, we introduce a *hybrid compile-time / runtime optimization* approach, as depicted by the red boxes in Figure 2. During compile-time, our goal is to find the optimal $\boldsymbol{\theta}_c^*$, by leveraging the correlation among all categories of parameters, to construct an ideal Spark context for query execution. Our compile-time optimization uses the cardinality estimates by Spark's cost-based optimizer.

During runtime, the Spark context remains fixed, and our runtime optimization runs as a plugin of AQE, invoked each time a new query stage is completed and a new collapsed query plan (denoted by $\overline{\text{LQP}}$) is generated. Our runtime optimization adjusts $\boldsymbol{\theta}_p$ for $\overline{\text{LQP}}$ based on the precise statistics of the completed stages. Then AQE applies $\boldsymbol{\theta}_p$ to its parametric rules, as well as non-parametric ones, to generate a new physical query plan ($\overline{\text{PQP}}$). As new query stages are produced in $\overline{\text{PQP}}$, our runtime optimization kicks in to optimize $\boldsymbol{\theta}_s$ parameters based on precise statistics. Then AQE applies parametric rules with the tuned $\boldsymbol{\theta}_s$, as well as non-parametric rules, to optimize data partitions of these stages.

3.3.2 Multi-Objective Optimization. Targeting cloud use, we cast our parameter tuning problem in the setting of multi-objective optimization where the objectives can be query latency, IO cost, and cloud cost in terms of CPU hours, memory hours, or a weighted combination of CPU, memory, and IO resources. It subsumes the solution of single-objective optimization and offers a theoretically sound approach to adapting to user preferences.

Formally, a multi-objective optimization (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).

$$\begin{aligned} \arg\min_{\boldsymbol{\theta}} \quad \boldsymbol{f}(\boldsymbol{\theta}) = \quad & [f_1(\boldsymbol{\theta}), f_2(\boldsymbol{\theta}), \ldots, f_k(\boldsymbol{\theta})] \\ & \boldsymbol{\theta} \in \Sigma \subseteq \mathbb{R}^d \\ s.t. \quad & \boldsymbol{f}(\boldsymbol{\theta}) \in \Phi \subseteq \mathbb{R}^k \\ & L_i \leq f_i(\boldsymbol{\theta}) \leq U_i, \quad i = 1, ..., k \end{aligned}$$

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. In general, the MOO problem leads to a set of solutions rather than a single optimal solution.

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 $\forall i \in [1, k], F_i' \leq F_i''$ 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 an example Pareto front in the 2D space of query latency and cloud cost. We make a few observations: First, 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. Second, the Pareto optimal points themselves represent tradeoffs between the two competing objectives: if the user desires lower latency, a higher cloud cost will be incurred, and vice-versa. The optimizer can recommend one Pareto solution 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 [41] of the Pareto points to the Utopia point $\boldsymbol{U}$,

which is the hypothetical (yet unattainable) optimum in all objectives, marked by the orange dot in Figure 4. For example, we can apply WUN to the Pareto set with the weight vector $\boldsymbol{w} = [0.9, 0.1]$ for peak hours and return the one that minimizes the weighted distance to the Utopia point. A few WUN recommendations for different weight vectors are shown in Figure 4.

Furthermore, prior work [23, 59, 66] used fixed weights to combine multiple objectives into a **single objective** (SO) and solve it to return one solution, denoted as the SO-FW method. Note that solving such a SO problem is different from computing the Pareto set $\{\boldsymbol{F_i}\}$ and then returning one solution from them using WUN:

$$\text{(SO:)} \arg\min_{\boldsymbol{\theta}} \boldsymbol{w} \cdot \boldsymbol{f}(\boldsymbol{\theta}) \neq \text{(WUN on Pareto Set:)} \arg\min_{F_1,\dots,F_i,\dots} \boldsymbol{w} \cdot ||\boldsymbol{F_i} - \boldsymbol{U}||$$

In fact, one classical MOO algorithm is *weighted sum* (WS) [30] that repeatedly applies different weight vectors to create a set of SO problems and returns all of their solutions, denoted as the MO-WS method, which subsumes SO-FW. It is known from the theory of WS that (1) each solution to a SO problem is Pareto optimal, but (2) 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 specific shape [30]. Figure 4 demonstrates that MO-WS gives poor coverage of the Pareto front: for TPCH-Q2, 11 SO problems generated from evenly spaced weight vectors return only two distinct solutions (marked by the blue dots): 10 out of 11 SO problems lead to the same bottom point, hence offering poor adaptability to the user preference. Increasing to 101 weight vectors still returns only 3 distinct points. In contrast, our MOO algorithm can offer a better-constructed Pareto front (the red line) at a lower time cost, hence better adaptability.

We next define the MOO problem for Spark parameter tuning.

Definition 3.3. Multi-Objective Optimization for Spark SQL

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

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

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

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 [15], as a representative of resource optimization solutions, 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 [65, 66] and recent Spark tuning systems [24, 25, 41, 55] cover mixed parameter space but do not support adaptive runtime optimization or multi-granularity tuning. The only system that supports MOO is UDAO [41] 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 a theoretically sound multi-objective optimization approach.

Table 2: Comparison of Spark parameter tuning methods

	Mixed Param. Space	Adaptive Runtime Opt.	Multi-Granularity	Multi-Objective
ReLocag [15]	×	×	×	×
BestConfig [66]	✓	×	×	×
ClassyTune [65]	✓	×	×	×
LITE[25]	✓	×	×	×
LOCAT [55]	✓	×	×	×
Li et. al [24]	✓	×	×	×
UDAO [41]	✓	×	×	✓
Ours	✓	✓	✓	✓

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 Multi-Granularity Support

The Spark optimizer offers the collapsed logical query plan ($\overline{\text{LQP}}$) and query stages in the physical plan to enable fine-grained tuning at runtime, but it does not provide any data structures for fine-grained tuning at compile time. Therefore, we introduce the notion of *subQ*, denoting a group of logical operators that will correspond to a query stage (QS) when the logical plan is translated to a physical plan, and build a model for each subQ. Hypothetically, a subQ will be transformed to a QS when it involves a scan operation or resides on the edge of a $\overline{\text{LQP}}$ with dependency cleared (i.e., when its preceding stages have completed). Since a physical query plan is divided into a directed acyclic graph (DAG) of QSs, we also consider the logical query plan (LQP) as a DAG of subQs at compile time, and treat the subQ as the finest unit for compile-time optimization. Figure 1(b) illustrates the LQP of TPCH-Q3, which can be divided into five subQs, each corresponding to one QS. To support optimization at runtime, we further construct performance models for the collapsed logical query plan ($\overline{\text{LQP}}$) and query stages (QS) accordingly.

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 using container technology with Spark's cluster manager, which ensures a dedicated allocation of cores and memory to the entire query. Such resource isolation enhances the

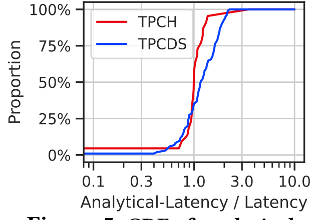


Figure 5: CDF of analytical latency over actual latency

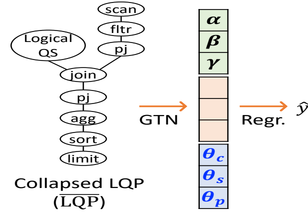


Figure 6: Model structure (GTN embedder + regressor) for $\overline{\text{LQP}}$

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 stages (subQs or QSs), leading to two challenges 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 research [28] assumed ample resources in industry-scale clusters and simplified this issue by taking the `max` latency among parallel tasks, but this approach 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 approach yields two significant advantages. Firstly, it establishes a direct link between the latency of a query and its constituent stages, enabling the computation of query-level latency at compile time through a `sum` aggregator over the task latencies of all subQs. Secondly, 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 analytical latency, we conducted a comparison with actual latencies at the query level using the TPC-H and TPC-DS benchmarks under the default Spark configuration. The results demonstrate a robust correlation between analytical and actual latencies, evidenced by Pearson correlation coefficients of 97.2% for TPC-H and 87.6% for TPC-DS. Furthermore, the distribution of the ratio between analytical and actual latencies, as illustrated in Figure 5, shows a predominant cluster around the value of 1. This indicates that analytical latency is not only a reliable predictor of actual query latency but also closely mirrors the actual execution time.

Cloud costs are primarily based on the consumption of resources, such as CPU-hour, memory-hour, and IO operations. Therefore, besides predicting latency and capturing CPU and memory usage from the context parameters $\boldsymbol{\theta_c}$, we also model IO operations as an additional factor. The cost for a query, similar to latency, can then be estimated by employing a `sum` aggregator from all subQs.

To summarize, our modeling work seeks to capture (1) end-to-end latency and cost for collapsed query plans ($\overline{\text{LQP}}$) at runtime, and (2) analytical latency and cost for subQs at compile time, as well as for query stages (QSs) at runtime (in the face of resource sharing), ensuring 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 the characteristics of queries and the dynamics of their execution environment, configured by decision variables and non-decision variables. First, we extract the query plan as a DAG of vectors, where each query operator is characterized through a composite encoding that integrates i) the operator type via one-hot encoding, ii) its cardinality, represented by row count and size in bytes, and iii) an average of the word embeddings [35] computed 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, encapsulating the statistics of the parallel-running stages in a numeric vector, tracking the number of their tasks in running and waiting states, and aggregating statistics of their finished task durations to characterize their behaviors. Lastly, 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 encoding, and other tabular features, poses a challenge in model formulation. To tackle this, we adopt a multi-channel input framework [28] that incorporates a Graph Transformer Network (GTN) [7] and a regressor to predict our objectives, as shown in Figure 6. We first derive the query embedding using a GTN model [7], which can handle the non-linear and non-sequential relationships and employ attention mechanisms and Laplacian positional encoding to capture operator correlations as well as positional information. These embeddings are then concatenated with other tabular data and processed through a regressor, allowing us to capture the interplay among the query characteristics, critical contextual factors, and tunable parameters.

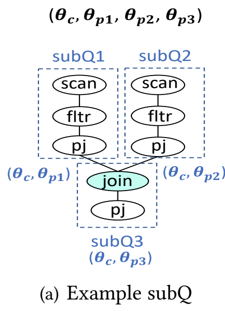
Adapting to Different Modeling Targets. Figure 6 illustrates the architecture of the $\overline{\text{LQP}}$ model, which has the largest number of all feature factors. For subQ models built at compile time, we adapt non-decision variables by deriving data characteristics from the cost-based optimizer ($\alpha = \alpha_{cbo}$), assuming uniform data distribution ($\beta = \vec{0}$) and the absence of resource contention ($\gamma = \vec{0}$). For the runtime QS model, we build a model by (1) updating runtime statistics as we described above, encoding the operators from the physical query plan, and dropping the $\boldsymbol{\theta_p}$ parameters as they have already been determined.

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. Our approach does so by exploring the correlation of $\boldsymbol{\theta_c}$ with fine-grained $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ parameters for different subqueries, under the modeling constraint that the non-decision



(a) Example subQ

1) **subQ tuning:** get the subQ-level MOO solutions F_s

subQ1 -- MOO

F_s	θ_c	θ_p	Lat	$
F^1_{s1}	θ^1_c	θ^1_{p1}	5	0.05
F^2_{s1}	θ^1_c	θ^2_{p1}	7	0.04
F^3_{s1}	θ^2_c	θ^3_{p1}	8	0.045

subQ2 -- MOO

F_s	θ_c	θ_p	Lat	$
F^1_{s2}	θ^1_c	θ^1_{p2}	9	0.015
F^2_{s2}	θ^2_c	θ^2_{p2}	4	0.022
F^3_{s2}	θ^2_c	θ^3_{p2}	10	0.013

subQ3 -- MOO

F_s	θ_c	θ_p	Lat	$
F^1_{s3}	θ^1_c	θ^1_{p3}	3	0.028
F^2_{s3}	θ^2_c	θ^2_{p3}	6	0.012

2) **DAG Aggregation:** get the query-level MOO solutions F_q

F_q	Θ	Lat	$
F^1_q	$\theta^1_c, \theta^1_{p1}, \theta^1_{p2}, \theta^1_{p3}$	17	0.093
F^2_q	$\theta^2_c, \theta^3_{p1}, \theta^2_{p2}, \theta^2_{p3}$	18	0.079
F^3_q	$\theta^2_c, \theta^3_{p1}, \theta^3_{p2}, \theta^2_{p3}$	24	0.07

3) **WUN**

$\Theta^2 = [\theta^2_c, \theta^3_{p1}, \theta^2_{p2}, \theta^2_{p3}]$
$F^2_q = [18, 0.079]$

Default Configuration $\Theta^4 = [\theta^1_c, \theta^2_{p1}, \theta^1_{p2}, \theta^1_{p3}]$ $F^4_q = [19, 0.083]$

Query Latency (s) / Cloud Cost ($): Default Configuration, Dominates, Recommended

(b) Approach Overview

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

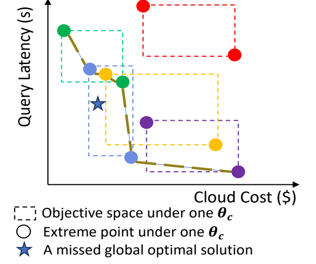


Figure 8: Boundary approximation of DAG optimization

variables for cardinality estimates are based on Spark's cost-based optimizer. Nevertheless, even under the modeling constraint, capturing the correlation between the mixed parameter space allows us to find a better Spark context for query execution, as we will demonstrate in our experimental study.

The multi-objective optimization problem in Def. 3.3 provides fine-grained control of $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$, at the subquery (subQ) level and query stage level, respectively, besides the query level control of $\boldsymbol{\theta_c}$. As such, the dimensionality of the parameter space is $d_c + m \cdot (d_p + d_s)$, where d_c, d_p and d_s denote the dimensionality of the $\boldsymbol{\theta_c}$, $\boldsymbol{\theta_p}$, $\boldsymbol{\theta_s}$ parameters, respectively, and m is the number of query stages. Such high dimensionality defeats most existing MOO methods when the solving time must be kept under the constraint of 1-2 seconds for cloud use, as we will show in performance evaluation.

To combat the high-dimensionality of the parameter space, we propose a new approach named *Hierarchical MOO with Constraints* (HMOOC). In a nutshell, it follows a divide-and-conquer framework to break a large optimization problem on $(\boldsymbol{\theta_c}, \{\boldsymbol{\theta_p}\}, \{\boldsymbol{\theta_s}\})$ to a set of smaller problems on $(\boldsymbol{\theta_c}, \boldsymbol{\theta_p}, \boldsymbol{\theta_s})$, one for subQ of the logical query plan (as defined in the previous section). 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)

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

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

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.

The main idea behind our approach is to tune each subQ independently under the constraint that $\boldsymbol{\theta_c}$ is identical among all subQ's.

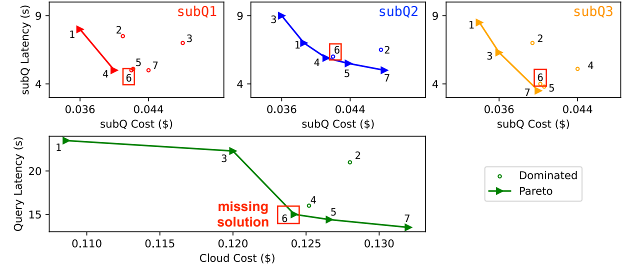


Figure 9: Example of missed global optimal solutions in TPCH Q3

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 $(\boldsymbol{\theta^1_c}, \boldsymbol{\theta^2_c})$ 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 subQ while obeying the constraint that all the subQs share the same $\boldsymbol{\theta_c}$. For simplicity, we focus on $(\boldsymbol{\theta_c}, \boldsymbol{\theta_p})$ in the following discussion as $\boldsymbol{\theta_s}$ is treated the same way as $\boldsymbol{\theta_p}$.

One may wonder whether it is sufficient to generate only the local Pareto solutions of $(\boldsymbol{\theta_c}, \boldsymbol{\theta_p})$ of each subQ. Unfortunately, this will lead to missed global Pareto optimal solutions due to the constraint on $\boldsymbol{\theta_c}$. Figure 9 illustrates an example with 3 subQs, where solutions sharing the same index fall under the same $\boldsymbol{\theta_c}$ configuration and have achieved optimal $\boldsymbol{\theta_p}$ under that $\boldsymbol{\theta_c}$ value. The first row in Figure 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 corresponding query-level values, where both query-level latency and cost are

the sums of subQ-level latency and cost. Notably, solution 6 is absent from the local subQ-level Pareto optimal solutions across all subQs. Due to the identical $\boldsymbol{\theta_c}$ constraint and the sum aggregation from subQ-level values to query-level values, although solution 6 is dominated in all subQs, the sum of its subQ-level latency and cost performs better than solution 4 (constructed from subQ-level Pareto optimal solutions) and is a query-level Pareto point.

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

1. Enriching $\boldsymbol{\theta_c}$ Candidates. To minimize the chance of missing global solutions, our first technique aims to preserve a diverse set of $\boldsymbol{\theta_c}$ configurations to be considered across all subQs. $\boldsymbol{\theta_c}$ can be initialized by random sampling or grid-search over its domain of values. Then, we employ a number of methods to enrich further the $\boldsymbol{\theta_c}$ set. In the case that the $\boldsymbol{\theta_c}$ values are initially randomly sampled, we draw inspiration from the evolutionary algorithms [8] and introduce a *crossover* operation over the existing $\boldsymbol{\theta_c}$ population to generate new candidates. If grid search is used to generate the initial $\boldsymbol{\theta_c}$ candidates, then we add random sampling to discover other values other than those covered in the grid search.

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

Proposition 5.1. Under any specific value $\boldsymbol{\theta_c^j}$, only subQ-level Pareto optimal solutions $(\boldsymbol{\theta_c^j}, \boldsymbol{\theta_p^*})$ contribute to the query-level Pareto optimal solutions.

In the interest of the space, all the proofs in this paper are deferred to Appendix A.1.

The above result allows us to restrict our search of $\boldsymbol{\theta_p}$ to only the local Pareto optimal ones. However, given the large, diverse set of $\boldsymbol{\theta_c}$ candidates, it is computationally expensive to solve the MOO problem for $\boldsymbol{\theta_p}$ repeatedly, once for each $\boldsymbol{\theta_c}$ candidate. We next introduce a clustering-based approximation to reduce the computation complexity. It is based on the hypothesis that, within the same subQ, similar $\boldsymbol{\theta_c}$ candidates entail similar optimal $\boldsymbol{\theta_p}$ values in the tuning process. By clustering similar $\boldsymbol{\theta_c}$ values into a small number of groups (based on their Euclidean distance), we then solve the MOO problem of $\boldsymbol{\theta_p}$ for a single $\boldsymbol{\theta_c}$ representative of each group. To expedite the repeated solving of $\boldsymbol{\theta_p}$ for different $\boldsymbol{\theta_c}$ representatives, we maintain a pool of samples of $\boldsymbol{\theta_p}$ and among them find the Pareto optimal values for each $\boldsymbol{\theta_c}$ representative. We then use the optimized $\boldsymbol{\theta_p}$ as the estimated optimal solution for other $\boldsymbol{\theta_c}$ candidates within each group.

Algorithm. Algorithm 1 describes the steps for obtaining an effective solution set of $(\boldsymbol{\theta_c}, \boldsymbol{\theta_p})$ for each subQ. Line 1 initiates the process by generating the initial $\boldsymbol{\theta_c}$ candidates, e.g. random sampling or grid-search. These candidates are then grouped using a clustering approach, where rep_c_list constitutes the list of $\boldsymbol{\theta_c}$ representatives for the n groups, C_list includes the members within all n groups, and κ represents the clustering model. In Line 3, $\boldsymbol{\theta_p}$ optimization is performed for each representative $\boldsymbol{\theta_c}$ candidate. Subsequently, the optimal $\boldsymbol{\theta_p}$ of the representative $\boldsymbol{\theta_c}$ is assigned to all members within the same group and is fed to the predictive models to get objective values (Line 4). After that, the initial effective set is obtained, where $\Omega^{(0)}$ represents the subQ-level objective values under different $\boldsymbol{\theta_c}$, and $\Theta^{(0)}$ represents the corresponding configurations. Line 5 further enriches $\boldsymbol{\theta_c}$ by either random sampling or applying our crossover method, which expands the initial effective set to generate new $\boldsymbol{\theta_c}$ candidates. Afterwards, the cluster model κ assigns the new $\boldsymbol{\theta_c}$ candidates with their group labels (Line 6). The previous optimal $\boldsymbol{\theta_p}$ values are then assigned to the new members within the same group, resulting in their corresponding subQ-level values as the enriched set (Line 7). Finally, the initial set and the enriched set are combined as the final effective set of subQ tuning (Line 8).

Algorithm 1: Effective Set Generation

Require: $Q, \phi_i, n, \forall i \in [1, k], \alpha, \beta, \gamma.$
1: $\boldsymbol{\Theta_c^{(0)}}$ = init_c()
2: $rep_c_list, C_list, \kappa$ = cluster($\boldsymbol{\Theta_c^{(0)}}, n$)
3: $\boldsymbol{\Theta_p^*}$ = optimize_p_moo($rep_c_list, \phi, \alpha, \beta, \gamma, Q$)
4: $\boldsymbol{\Omega^{(0)}}, \boldsymbol{\Theta^{(0)}}$ = assign_opt_p($C_list, rep_c_list, \boldsymbol{\Theta_p^*}, \phi, \alpha, \beta, \gamma, Q$)
5: $\boldsymbol{\Theta_c^{(new)}}$ = enrich_c($\Omega^{(0)}, \Theta^{(0)}$)
6: $C_list^{(new)}$ = assign_cluster($\boldsymbol{\Theta_c^{(new)}}, rep_c_list, \kappa$)
7: $\boldsymbol{\Omega^{(new)}}, \boldsymbol{\Theta^{(new)}}$ = assign_opt_p($C_list^{(new)}, rep_c_list, \boldsymbol{\Theta_p^*}, \phi, \alpha, \beta, \gamma, Q$)
8: Ω, Θ = union($\Omega^{(0)}, \Theta^{(0)}, \Omega^{(new)}, \Theta^{(new)}$)
9: **return** Ω, Θ

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

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

HMOOC1: Divide-and-Conquer. Under a fixed $\boldsymbol{\theta_c}$, i.e., satisfying the constraint inherently, we propose a divide-and-conquer method to compute the Pareto set of the simplified DAG, which is reduced to a list of subQs. The idea is to (repeatedly) partition the list into two halves, solve their respective subproblems, and merge their solutions to global Pareto optimal ones. The merge operation enumerates all the combinations of solutions of the two subproblems, sums up their objective values, and retains only the Pareto optimal ones. Our proof (available in Appendix A.1) shows that this method returns a full set of query-level Pareto optimal solutions

as it enumerates those combinations of subQ-level solutions that have a chance to be global Pareto optimal.

HMOOC2: WS-based Approximation. We propose a second technique to approximate the MOO solution over a list structure. For each fixed $\boldsymbol{\theta_c}$, we apply the weighted sum (WS) method to generate evenly spaced weight vectors. Then for each weight vector, we obtain the (single) optimal solution for each subQ and sum the solutions of subQ's to get the query-level optimal solution. It can be proved that this WS method over a list of subQs guarantees to return a subset of query-level Pareto solutions. Further details of this method are deferred to Appendix A.1).

HMOOC3: Boundary-based Approximation. Given that DAG aggregation under each $\boldsymbol{\theta_c}$ candidate operates independently, it is inefficient to do so repeatedly when we have a large number of $\boldsymbol{\theta_c}$ candidates. Our next approximate technique stems from the idea that the objective space of DAG aggregation under each $\boldsymbol{\theta_c}$ can be approximated by k *extreme points*, where k is the number of objectives. In this context, the *extreme point* under a fixed $\boldsymbol{\theta_c}$ is the Pareto optimal point with the best query-level value for any objective. Then, the approximate query-level Pareto set is determined by the non-dominated extreme points among all $\boldsymbol{\theta_c}$ points.

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

Therefore, we consider the degenerated *extreme points* to symbolize the 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 $\boldsymbol{\theta_c}$ candidates. The brown dashed line represents the approximate query-level Pareto front derived by filtering the dominated solutions from the collection of extreme points. The star solution indicates a missed query-level Pareto solution, as it cannot be captured from the extreme points.

The algorithm works as follows. For each $\boldsymbol{\theta_c}$ candidate, for each objective, we select the subQ-level solution with the best value for that objective for each subQ, and then sum up the objective values of such solutions from all subQs to form one 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 $\boldsymbol{\theta_c}$ candidates. An additional filtering step will retain the non-dominated solutions from the kn candidates, using an existing method of complexity $O(kn\log(kn))$ [20].

Our formal results include the following:

Proposition 5.2. Under a fixed $\boldsymbol{\theta_c}$ candidate, the query-level objective space of Pareto optimal solutions is bounded by its extreme points in a 2D objective space.

Proposition 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.

5.2 Runtime Optimization

While the compile-time optimization provides a fine-grained configuration of the parameters, it relies on estimated cardinality and assumption of uniform data distribution and no resource contention. In addition, Spark accepts only one copy of $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ at the query submission time and can change the physical query plan during the AQE. Therefore, the true value of compile-time optimization is to recommend the optimal context parameters $\boldsymbol{\theta_c^*}$ by considering the correlations with $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$. Then, our runtime optimization addresses the remaining problems, adapting $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ based on actual runtime statistics and plan structures.

Given the constraint that Spark takes only one copy of $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ at query submission time, we intelligently aggregate the fine-grained $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ 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 (as shown in Figure 3(b)). To mitigate this, we initialize $\boldsymbol{\theta_p}$ with the smallest threshold among all join-based subQs, enabling more informed runtime decisions. Other details of aggregating $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ are in Appendix A.3.

Runtime optimization 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 machine learning models and MOO algorithms [41], 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, TPCDS 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 Appendix A.3. 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 $\boldsymbol{\theta_c}$, 9 for $\boldsymbol{\theta_p}$, and 2 for $\boldsymbol{\theta_s}$, selected based on feature selection profiling [18] and best practices from the Spark documentation. More details are in Appendix B.1.

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 [32].

Table 3: Model performance with Graph+Regressor

	Target	Ana-Latency/Latency (s) WMAPE	P50	P90	Corr	IO (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

Table 4: Latency reduction with a strong speed preference

	TPC-H MO-WS	HMOOC3	HMOOC3+	TPC-DS MO-WS	HMOOC3	HMOOC3+
Coverage (1s)	5%	95%	68%	0%	98%	96%
Coverage (2s)	36%	100%	100%	0%	100%	100%
Total Lat Reduction	18%	59%	61%	25%	59%	64%
Avg Lat Reduction	-1%	52%	52%	34%	54%	57%
Avg Solving Time (s)	**2.6**	**0.52**	**0.83**	**15**	**0.47**	**0.62**
Max Solving Time (s)	4.5	1.01	1.55	68	1.24	1.34
Avg $\frac{\text{Lat Reduction}}{\text{Solving Time}}$	**1%**	**103%**	**71%**	**3%**	**127%**	**99%**

6.1 Model Evaluation

We trained separate models for subQ, QS, and $\overline{\text{LQP}}$ to support compile-time/runtime optimization. The traces of each workload were split into 8:1:1 for training, validation, and testing. We conducted hyperparameter tuning in a GPU node with 4 NVIDIA A100 cards. We evaluate each model with the following metrics: 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 TPCH and TPCDS in 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, IO is more predictable than latency, evidenced by a WMAPE of 0.2-11% and almost perfect 99-100% correlation with the actual IO, 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. Overall, these results demonstrate the robust performance of our models in predicting cost performance metrics for Spark queries, while enabling efficient optimization.

Expt 2: *Comparison of Compile-time and Runtime Results.* We then look into the performance differences between the runtime QS and its corresponding subQ at compile time. First, the latency performance in runtime QS is slightly inferior to its corresponding subQ at compile time. This disparity can be attributed to the runtime QS's exposure to more varied and complex query graph structures, which complicates the prediction process. Second, the runtime QS consistently surpasses the subQ in IO prediction. This superior performance is linked to the direct correlation between IO and input size; the runtime QS benefits from access to actual input sizes, thereby facilitating more precise predictions. In contrast, the subQ must base its predictions on input sizes estimated by the cost-based optimizer (CBO), which introduces more errors.

6.2 Compile-time MOO Methods

We next evaluate our compile-time MOO methods for fine-grained tuning against SOTA MOO methods: the *weighted sum* (WS) [30], *evolutionary* (Evo) [8], and *progressive frontier* (PF) [41] methods, which were reported to be the most competitive methods [41].

Expt 5: *Comparison of DAG Aggregation methods.* Figures 10(a) and 10(b) compare the three DAG aggregation methods (§5.1.2) in the HMOOC framework using the two benchmarks. Hypervolume (HV) is a standard measure of the dominated space of a Pareto set in the objective space. As depicted in Figures 10(a) and 10(b), the three methods demonstrate similar HV but vary in time cost. The Boundary-based Approximation (HMOOC3) is the most efficient for both benchmarks without losing much HV, achieving the solving time of 0.32-1.72s (in particular, all under 1 second for TPCH). Therefore, we use HMOOC3 in the remaining experiments.

Expt 6: *Comparison with SOTA MOO methods.* We next compare HMOOC3 with 3 SOTA MOO methods, WS [30] (with tuned hyperparameters of 10k samples and 11 pairs of weights), Evo [8] (with a population size of 100 and 500 function evaluations), and PF [41], for fine-grained tuning of parameters based on Def. 3.3.

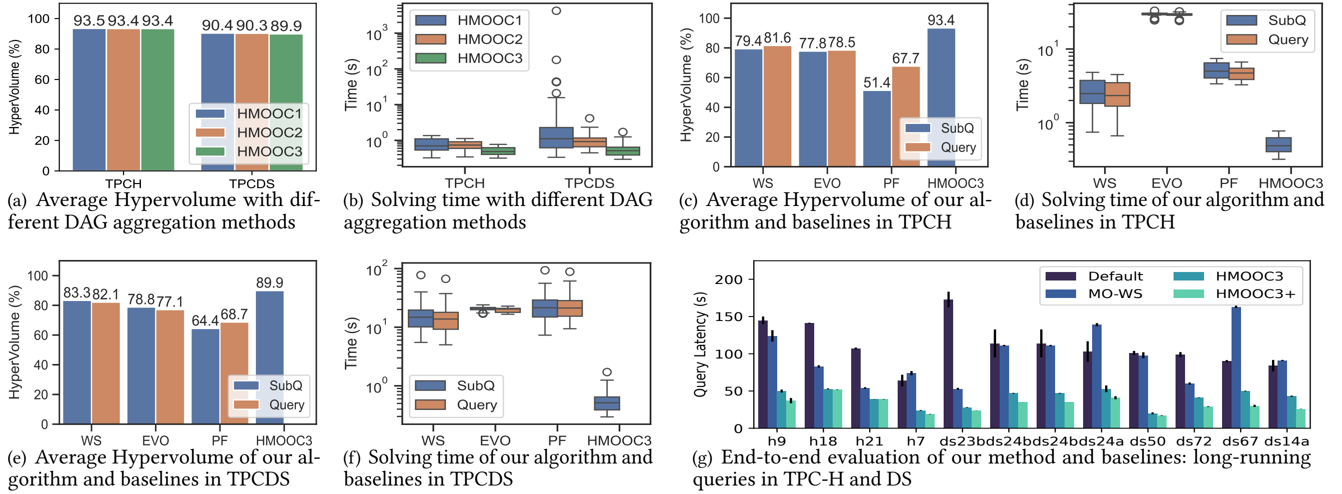
Figures 10(c)-10(e) compare the methods in HV and solving time, where the blue bars represent fine-grained tuning. HMOOC3 achieves the highest average HV among all methods, 93.4% in TPCH and 89.9% in TPCDS, and the lowest time cost, within 0.5-0.55s. It outperforms other methods with 7.9%-81.7% improvement in HV and 81.8%-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, which includes one set of $\boldsymbol{\theta_c}$ and m sets of $\boldsymbol{\theta_p}$, where m is the number of subQs in a query.

Expt 7: *Comparison with query-level tuning*. We next consider WS, EVO, and PF only for coarse-grained, query-level tuning, which sets one copy of $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ values for uniform control of all subQs, to reduce its parameter space. The orange bars in Figures 10(c)- 10(f) depict the performance under query-control tuning. For TPCH, both the HV and solving time under query-level tuning perform better than those under subQ-level tuning, due to a much smaller search space. But it still takes over an average of at least 2.3s and much lower HV (at most 81.6%) than HMOOC3 (93.4%). For TPCDS, except PF, WS and EVO perform slightly worse in HV, and they all improve solving time somewhat. However, all of them lose to HMOOC3 in HV (at most 83.3% v.s. 89.9%) and in solving time (the average exceeding 10s v.s. 0.55s).

6.3 End-to-End Evaluation

We integrate runtime optimization with the best-performing compile-time optimization method `HMOOC3`, denoted as `HMOOC3+`, and 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 9: *End-to-end benefits against query-level MOO.* We first show the advantages of our methods (`HMOOC3` and `HMOOC3+`) in reducing latency compared to the best-performing MOO method from



(a) Average Hypervolume with different DAG aggregation methods

(b) Solving time with different DAG aggregation methods

(c) Average Hypervolume of our algorithm and baselines in TPCH

(d) Solving time of our algorithm and baselines in TPCH

(e) Average Hypervolume of our algorithm and baselines in TPCDS

(f) Solving time of our algorithm and baselines in TPCDS

(g) End-to-end evaluation of our method and baselines: long-running queries in TPC-H and DS

Figure 10: Analytical and end-to-end performance of our algorithm, compared to the state-of-the-art (SOTA) methods

Table 5: Latency and cost adapting to preferences

Prefs. Lat/Cost	TPC-H		TPC-DS	
	SO-FW	HMOOC3+	SO-FW	HMOOC3+
(0.0, 1.0)	20% / -11%	-17% / -9%	-6% / 64%	-47% / -22%
(0.1, 0.9)	1% / 1%	-25% / -5%	-28% / 105%	-51% / -12%
(0.5, 0.5)	-1% / 25%	-43% / 2%	-28% / 128%	-57% / 16%
(0.9, 0.1)	-13% / 27%	-52% / 9%	-34% / 139%	-57% / 45%
(1.0, 0.0)	-14% / 44%	-52% / 12%	-26% / 144%	-58% / 50%

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 4 show the improvement over the default Spark configuration. First, *fine-grained tuning significantly enhances performance* (labeled as major result **R1**), cutting latency by 59% for both benchmarks with compile-time optimization (HMOOC3), and by 61% and 64% for TPC-H and TPC-DS, respectively, further with runtime optimization (HMOOC3+). They both outperform MO-WS with only 18-25% reductions and in some cases, underperforming the default configuration. Second, MO-WS, *even limited to query-level tuning, suffers in efficiency*, finding Pareto solutions for merely 36% and 0% of queries within 2s in TPCH and TPCDS, respectively. In contrast, our approach solves MOO for all queries with an average time of 0.62-0.83s and a maximum of 1.34-1.55s. When we consider a new efficiency measure, defined as latency reduction per unit of solving time, our method vastly outperforms MO-WS, *achieving a 1-2 order-of-magnitude improvement in efficiency for latency reduction* (**R2**).

Further, when runtime optimization is enabled, HMOOC3+ *enables more benefits for longer-running queries* (**R3**), which are often complex to optimize and suffer from suboptimal query plans with the static θ_p tuned based on the cardinality estimates and simplifying assumptions at compile time. Figure 10(g) shows such reductions for the long-running queries from TPCH and TPCDS. Compared to HMOOC3, HMOOC3+ achieves up to a 22% additional latency reduction over the default configuration. More details are in Appendix B.3.

Expt 10: *Adaptability Comparison to SO with fixed weights.* As MO-WS is not practical for cloud use due to its inefficiency, we now compare the adaptability of our approach against the most common, practical approach that combines multiple objectives into a single objective using fixed weights [23, 59, 66], denoted as SO-FW. The evaluation, presented in Table 5, focuses on the average reduction rates in latency and cost relative to the default configurations across a range of preference vectors for TPCH and TPCDS queries. First, HMOOC3+ *dominates* SO-FW *with more latency and cost reductions in most cases* (**R4**), achieving up to 52-64% latency reduction and 9-22% 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* (**R5**), enhancing latency reductions progressively as preferences shift towards speed. In contrast, SO-FW does not make meaningful recommendations. Specifically, under a cost-saving preference of $(0.0, 1.0)$, SO-FW struggles to lower costs in TPCDS, 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 47% 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 parameter tuning that achieves multi-granularity 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 61% and 64% latency reduction on average for TPC-H and TPC-DS, respectively, under the solving time of 0.62-0.83 sec, outperforming the most competitive MOO method with 18-25% latency reduction and high solving time of 2.4-15 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 and heterogeneous clusters in cloud deployment.

REFERENCES

[1] 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

[2] David Arthur and Sergei Vassilvitskii. 2006. How slow is the k-means method?. In *Proceedings of the twenty-second annual symposium on Computational geometry*. 144–153.

[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] 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

[5] 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.

[6] 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

[7] 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).

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

[9] 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.

[10] 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

[11] 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.

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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.

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

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

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

[32] 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

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

[34] 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.

[35] 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

[36] 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

[37] 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.

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] 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.

[44] 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.

[45] 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

[46] 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

[47] 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

[48] 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

[49] 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

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

[51] 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

[52] 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

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

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

[55] 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

[56] 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

[57] 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

[58] 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

[59] 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

[60] 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

[61] 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

[62] 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

[63] 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

[64] 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

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

[66] 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.

Algorithm 2: General_Divide_and_conquer

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

Algorithm 3: merge

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

A ADDITIONAL MATERIALS FOR MOO

A.1 Algorithms and Proofs

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

A.1.1 Subquery (subQ) Tuning. Below, we provide the proof of Proposition 5.1.

Proposition 5.1 Under any specific value $\boldsymbol{\theta_c^j}$, only subQ-level Pareto optimal solutions $(\boldsymbol{\theta_c^j}, \boldsymbol{\theta_{pi}^*})$ for the i-th subQ contribute to the query-level Pareto optimal solutions $(\boldsymbol{\theta_c^j}, \{\boldsymbol{\theta_p^*}\})$.

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

We have

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

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

Complexity Analysis of Algorithm 1. The complexity of the *cluster* function depends on the choice of the cluster algorithm. For example, given N initialized $\boldsymbol{\theta_c}$ candidates, the average complexity of k-means is $O(C \cdot N \cdot T)$, where N is the number of samples for $\boldsymbol{\theta_c}$, C is the number of clusters, and T is the number of iterations [2]. The *optimize_p_moo* function solves MOO problems that optimize $\boldsymbol{\theta_p}$ under different $\boldsymbol{\theta_c}$ representatives of all subQs. Assuming optimizing $\boldsymbol{\theta_p}$ under a fixed $\boldsymbol{\theta_c}$ of each subQ is λ, the complexity of the *optimize_p_moo* function is $O(m \cdot C \cdot \lambda)$, where m is the number of subQs. The *assign_opt_p* function assigns optimal $\boldsymbol{\theta_p}$ of the representative $\boldsymbol{\theta_c}$ to all members within the same $\boldsymbol{\theta_c}$ group. Given the average number of optimal $\boldsymbol{\theta_p}$ among all $\boldsymbol{\theta_c}$ as p_{avg}, for m subQs, the complexity of the *assign_opt_p* function is $O(p_{\text{avg}} \cdot N \cdot m)$. The *enrich_c* function either random samples $\boldsymbol{\theta_c}$ or applying our heuristic method, which includes two steps. The first step is to union the local optimal $\boldsymbol{\theta_c}$ of all subQs, where filtering dominated solutions of m subQs takes $O(m \cdot (p_{\text{avg}} \cdot N) \cdot \log(p_{\text{avg}} \cdot N))$ [20]. And the second step addresses a Cartesian product of two lists. Given the lengths of two lists as N_1 and N_2 respectively, its complexity is $O(N_1 \cdot N_2)$. The *assign_cluster* function utilizes the previous cluster model to obtain cluster labels for new $\boldsymbol{\theta_c}$ candidates, whose complexity is linear with the number of new $\boldsymbol{\theta_c}$ candidates.

A.1.2 DAG Aggregation. We provide further details of three methods for DAG aggregation.

HMOOC1: Divide-and-Conquer. This DAG aggregation method is described in Algorithm 2. The Ω, Θ are the effective set of all subQs, where Ω represents subQ-level objective values, and Θ represents the corresponding configurations, including $\{\boldsymbol{\theta_c}, \{\boldsymbol{\theta_p}\}, \{\boldsymbol{\theta_s}\}\}$. If there is only one subQ, it returns the Ω, Θ (lines 1-2). Otherwise, it follows a divide-and-conquer framework (lines 4-8).

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

From the view of optimality, Algorithm 2 is proved to return a full set of the query-level Pareto optimal solutions as it enumerates over all subQ-level solutions. The complexity of *merge* function is $O(M*N) + O((M*N)log(M*N))$ if there are M and N solutions in two nodes, where the enumeration takes $O(M*N)$ and filtering dominated solutions takes $O((M*N)log(M*N))$. While after merging several times, M and N could be high. Thus the total complexity could be high.

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

Proposition A.1. Algorithm 2 always output the full Pareto front of the simplified DAG.

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

$$Pf(Pf(D) \oplus Pf(G)) = Pf(D \times G)$$

Let $E_D : C_D \to \mathbb{R}$ be the evaluation function of node D, where C_D is the set of configurations for node D. We define E_G in a similar

Algorithm 4: Compressing_list_nodes

```
Require: subQ_list, ws_pairs.
1: PO = [ ], conf = [ ]
2: for [w_l, w_c] in ws_pairs do
3:     po_n = [ ], conf_n = {}
4:     for subQ_po, subQ_conf in subQ_list do
5:         subQ_po_norm = normalize_per_subQ(subQ_po)
6:         opt_ind = minimize_ws([w_l, w_c], subQ_po_norm)
7:         po_n.append(subQ_po[opt_ind])
8:         conf_n.append(subQ_conf[opt_ind])
9:     end for
10:    PO.append(sum(po_n)), conf.append(conf_n)
11: end for
12: return filter_dominated(PO, conf)
```

manner. We also define $E : (c_D, c_G) \mapsto E_D(c_D) + E_G(c_G)$, where c_D and c_G are one configuration in C_D and C_G respectively.

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

$$p = p_D + p_G, \qquad p_D \in Pf(D), p_G \in Pf(G)$$

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

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

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

By combining the two inclusions, we obtain that $Pf(Pf(D) \oplus Pf(G)) = Pf(D \times G)$, where the left side is the Pareto optimal solution from the Algorithm 2 and the right side is the Pareto optimal solution over the whole configuration space of two nodes. Thus, Algorithm 2 returns a full set of Pareto solutions.

□

HMOOC2: WS-based Approximation. We propose a second technique to approximate the MOO solution over a list structure.

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

Algorithm 4 describes the full procedures. The input includes a subQ_list, which includes both subQ-level objective values and the corresponding configurations of all subQs. ws_paris are the weight pairs, e.g. $[[0.1, 0.9], [0.2, 0.8], \ldots]$ for latency and cost. Line 1 initializes the query-level objective values and configurations. Lines 3-9 address the Weighted Sum (WS) method to generate the query-level optimal solution for each weight pair. Specifically, Lines 5-8 apply the WS method to obtain the optimal solution choice for each subQ, and sum all subQ-level values to get the query-level values. Upon iterating through all weights, a Pareto solution set is derived after the necessary filtering (Line 12).

Theoretical Analysis

Lemma 1. For a DAG aggregation problem with k objectives that use the `sum` operator only, Algorithm 4 guarantees to find a non-empty subset of query-level Pareto optimal points under a specified $\boldsymbol{\theta_c}$ candidate.

In proving Lemma 1, we observe that Algorithm 4 is essentially a Weighted Sum procedure over Functions (WSF). Indeed we will prove the following two Propositions: 1) each solution returned by WSF is Pareto optimal; 2) the solution returned by the Algorithm 4 is equivalent to the solution returned by WSF. Then it follows that the solution returned by Algorithm 4 is Pareto optimal.

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

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

$$F_v = H_v(x; f) = \sum_{i=1}^{m} \sum_{j=1}^{p_i} x_{ij} \times f^v_{ij},$$

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

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

$$\underset{x}{\text{argmin}}(\sum_{v=1}^{k} w_v * H_v(x))$$

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

Where w_v is the weight value for objective v. Next, we prove for Lemma 1. As stated before, It is done in two steps.

Proposition A.2. The solution constructed using x returned by WSF is Pareto optimal.

Proof.

Assume that x^* (corresponding to $[F^*_1, ..., F^*_k]$) is the solution of WSF. Suppose that another solution $[F'_1, ..., F'_k]$ (corresponding

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

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

Proposition A.3. The optimal solution returned by the Algorithm 4 is equivalent to the solution constructed using x returned by WSF.

Proof.

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

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

For the solution $[F_1^{''}, ..., F_k^{''}]$ returned by Algorithm 4, x'' represents the corresponding selection. It is achieved by minimizing the following formula:

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

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

So, x'' must solve:

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

Here, optimizing for each subQ is independent of the optimization of the other subQs, so we can invert the sum over i and the arg min. Thus, $x' = x''$. Therefore, WSF and Algorithm 4 are equivalent. □

With these two propositions, we finish the proof of Lemma 1.

Algorithm 4 varies w weight vectors to generate multiple query-level solutions. And under each weight vector, it takes $O(m \cdot p_{max})$ to select the optimal solution for each LQP-subtree based on WS, where p_{max} is the maximum number of solutions among m subQs. Thus, the overall time complexity of one $\boldsymbol{\theta_c}$ candidate is $O(w \cdot (m \cdot p_{max}))$.

HMOOC3: Boundary-based Approximation.

We now show the proof of Proposition 5.2.

Proposition 5.2 Under a fixed $\boldsymbol{\theta_c}$ candidate, the query-level objective space of Pareto optimal solutions is bounded by its extreme points in a 2D objective space.

Proof.

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

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

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

We next show the proof of Proposition 5.3.

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.

Proof.

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

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

A.2 Compile-time optimization

A.2.1 θ_c enrichment. The following theoretical result sheds light on subQ tuning, indicating that $\boldsymbol{\theta_c}$ derived from the subQ-level Pareto optimal solutions can serve as a useful warm-start for generating new $\boldsymbol{\theta_c}$ configurations.

Proposition A.4. As shown in Figure 11, for all subQs, solutions with the same $\boldsymbol{\theta_c}$ located in the red region cannot contribute to query-level Pareto optimal solutions.

Proof. In any subQ s of an arbitrary DAG, the dominated solution f^s (red area) is dominated by f'^s with any arbitrary $\boldsymbol{\theta_c}$.

Given query-level solution F built from f^s, F' built from f'^s, supposing F is non-dominated with F' or dominates F', there must be at least one f^s with lower latency or cost than f'^s, which is impossible and concludes the proof. □

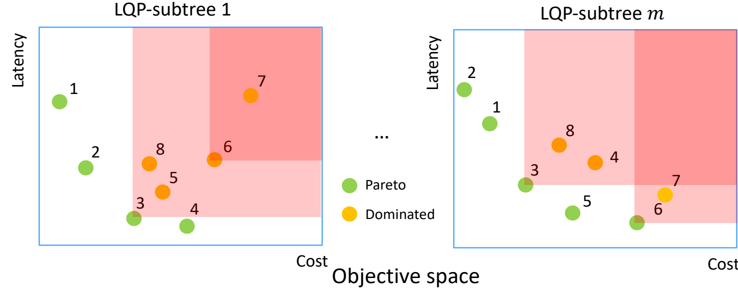


Figure 11: Objective space

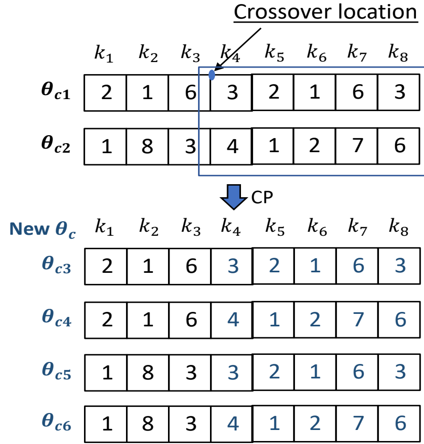


Figure 12: Example of generating new θ_c candidates

To uncover unexplored $\boldsymbol{\theta_c}$ configurations, drawing inspiration from the crossover operation in evolutionary algorithms, we introduce a heuristic method termed $\boldsymbol{\theta_c}$ *crossover*. The core concept involves utilizing the *Cartesian Product* (CP) operation to generate new $\boldsymbol{\theta_c}$ candidates from the existing pool. This operation is executed by randomly selecting a crossover location to divide the initial $\boldsymbol{\theta_c}$ configurations into two parts. The following example illustrates this process.

Example In Figure 12, $\boldsymbol{\theta_{c1}}$ and $\boldsymbol{\theta_{c2}}$ represent initial $\boldsymbol{\theta_c}$ candidates obtained from the subQ-level tuning, each consisting of 8 variables (e.g., $k_1, ..., k_8$). The values within the boxes denote the configurations of $\boldsymbol{\theta_c}$. The blue point denotes a randomly generated crossover location. Based on this location, $\boldsymbol{\theta_{c1}}$ and $\boldsymbol{\theta_{c2}}$ are divided into two parts: k_1, k_2, k_3 and k_4, k_5, k_6, k_7, k_8, delineated by a blue rectangle. A Cartesian Product (CP) is then applied to these two parts. In this example, there are two distinct configurations for k_1, k_2, k_3 and two distinct configurations for k_4, k_5, k_6, k_7, k_8. Consequently, the CP generates four configurations, represented as $\boldsymbol{\theta_{c3}}$ to $\boldsymbol{\theta_{c6}}$. It is noteworthy that $\boldsymbol{\theta_c}$ *crossover* generates new $\boldsymbol{\theta_c}$ configurations (e.g., $\boldsymbol{\theta_{c4}}$, $\boldsymbol{\theta_{c5}}$) without discarding the initial $\boldsymbol{\theta_c}$ candidates (e.g., $\boldsymbol{\theta_{c3}}$, $\boldsymbol{\theta_{c6}}$).

Analysis on Boundary-based Approximation

Since different $\boldsymbol{\theta_c}$ values result in diverse total resources and cover various regions of the Pareto frontier, it's noteworthy that all DAG optimization methods produce the same effective set, as confirmed by Figure 13. In scenarios with multiple $\boldsymbol{\theta_c}$ candidates, the boundary-based method achieves comparable hypervolume to the others.

Experiments

Analysis on Query-control

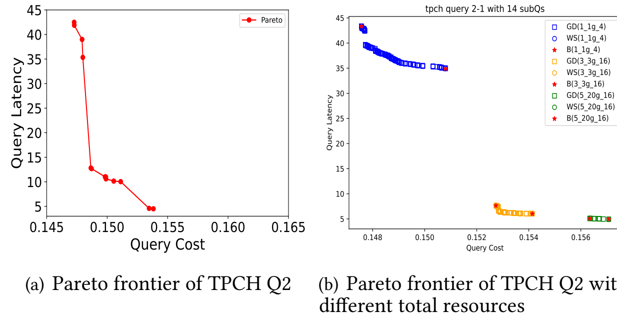
It's worth noting that query-control cannot achieve a higher upper-bound than the finer-control. To verify this, we implemented a smaller search space (each parameter having only 2 values) for WS to fully explore query-control, where WS performs the best among all baselines for both TPCH and TPCDS. Figure 14 displays the hypervolume of WS under different numbers of samples, with blue and orange bars representing hypervolume of finer-control and query-control, respectively. It is observed that as the number of samples increases, the hypervolume of query-control plateaus at 1M samples (89%), while the hypervolume of finer-control continues to improve (90%), illustrating the necessity of finer-control in our problem.

A.3 Additional Materials for Runtime Optimization

A.3.1 More on $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ Aggregation. Ideally, one could copy $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ from the initial subQ, allowing the runtime optimizer to adjust them by adapting to the real statistics.

Given the constraint that Spark takes only one copy of $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ at query submission time, we intelligently aggregate the fine-grained $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ 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 (as shown in Figure 3(b)). On the other hand, setting these thresholds to zero at SQL submission might overlook opportunities for applying BHJs, especially for joins rooted in scan-based subQs with small input sizes. To mitigate this, we initialize $\boldsymbol{\theta_p}$ with the smallest threshold among all join-based subQs, enabling more informed runtime decisions. Other details of aggregating $\boldsymbol{\theta_p}$ and $\boldsymbol{\theta_s}$ are in Appendix A.3. We also cap these thresholds (10MB for broadcast threshold and 0MB for shuffle hash threshold) at their default values to ensure BHJs are not missed for small scan-based subQs.

A.3.2 More on Pruning Optimization Requests. To address this, we established rules to prune unnecessary requests based on the



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

Figure 13: Objective space under differen resources

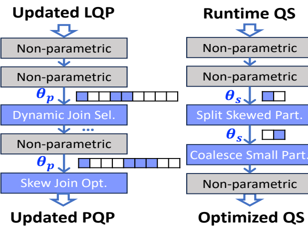


Figure 15: Runtime Opt. Rules

Table 6: (Selected) Spark parameters in three categories

θ_c	**Context Parameters**
k_1	`spark.executor.cores`
k_2	`spark.executor.memory`
k_3	`spark.executor.instances`
k_4	`spark.default.parallelism`
k_5	`spark.reducer.maxSizeInFlight`
k_6	`spark.shuffle.sort.bypassMergeThreshold`
k_7	`spark.shuffle.compress`
k_8	`spark.memory.fraction`
θ_p	**Logical Query Plan Parameters**
s_1	`spark.sql.adaptive.advisoryPartitionSizeInBytes`
s_2	`spark.sql.adaptive.nonEmptyPartitionRatioForBroadcastJoin`
s_3	`spark.sql.adaptive.maxShuffledHashJoinLocalMapThreshold`
s_4	`spark.sql.adaptive.autoBroadcastJoinThreshold`
s_5	`spark.sql.shuffle.partitions`
s_6	`spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes`
s_7	`spark.sql.adaptive.skewJoin.skewedPartitionFactor`
s_8	`spark.sql.files.maxPartitionBytes`
s_9	`spark.sql.files.openCostInBytes`
θ_s	**Query Stage Parameters**
s_{10}	`spark.sql.adaptive.rebalancePartitionsSmallPartitionFactor`
s_{11}	`spark.sql.adaptive.coalescePartitions.minPartitionSize`

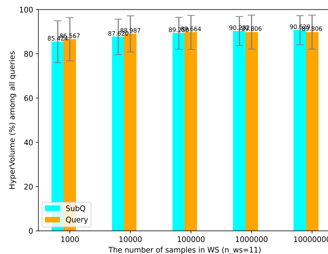


Figure 14: Comparison of query-control and finer-control with smaller searching space

runtime semantics of parametric rules: LQP parametric rules are used to decide join algorithms and QS parametric rules are used to re-balance the data partitions in a post-shuffle QS. Therefore, we bypass requests for non-join operations and defer requests for LQP containing join operators until all input statistics are available, thereby avoiding decisions based on inaccurate cardinality estimations. Additionally, we skip all the scan-based QSs and only send the requests when the input size of a QS is larger than the target partition size (configured by s_1). By applying the above rules, we substantially reduce the total number of optimization calls by 86% and 92% for TPCH and TPCDS respectively.

B ADDITIONAL EXPERIMENTAL DETAILS

B.1 More Setup

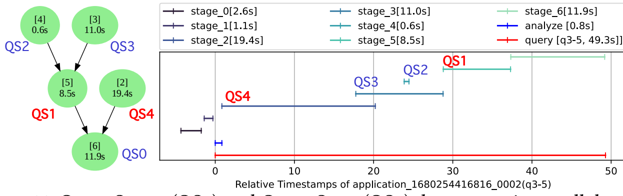
B.1.1 Hardware. We use two 6-node Spark 3.5.0 clusters with runtime optimization plugins. Each node is CentOS-based with 2 16-core Intel Xeon Gold 6130 processors, 768GG of RAM, and RAID disks, connected with 100Gbps Ethernet.

B.1.2 Parametric Optimization Rules. Figure 15 illustrates the transformations of a collapsed LQP and a runtime QS, which both pass through a pipeline of parametric rules (blue) and non-parametric rules (gray).

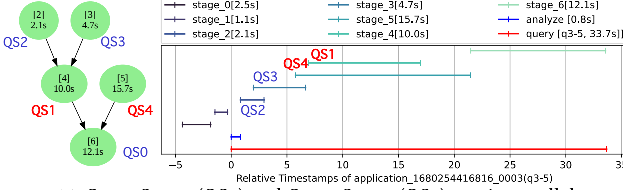
B.1.3 Spark Parameters Details. We list our 19 selected Spark parameters in Table 1, which are categorized into three groups: context parameters, logical query plan parameters, and query stage parameters. The default configuration is set to Spark's default values.

B.2 Specific Knob Concerns

spark.sql.adaptive.enable=true The parameter was introduced in Spark 1.6 and has been set to true by default since Spark 3.2. We have chosen to enable it for two main reasons. First, enabling adaptive query execution (AQE) allows us to perform parameter tuning at the stage level. Specifically, our runtime optimizer could fine-tune the runtime parameters (at the stage level) while AQE



(a) QueryStage1(QS1) and QueryStage(QS4) do not run in parallel



(b) QueryStage1(QS1) and QueryStage4(QS4) run in parallel

Figure 16: Repeated Runs of TPCH Q3 with the Same Configuration. In Figure 16(a), QS4 runs logically ahead of QS2 and QS3 and finishes before running QS1. Therefore, QS1 runs by itself without any resource sharing, and the query takes 49.3s. In Figure 16(b), QS2 and QS3 run logically ahead of QS4, and hence QS1 and QS4 run in parallel by sharing the resources with a 33.7s query latency.

re-optimizes the query plan based on the actual runtime statistics in the middle of query execution. Second, enabling AQE improves the robustness of query latency. When AQE is disabled, the DAGScheduler asynchronous converts the entire query to a DAG of stages. Consequently, parallel stages can be randomly interleaved during query execution, leading to unpredictable query latencies. For instance, Figure 16 demonstrates that disabling AQE can result in different stage interleaving patterns, causing a significant 46% increase in query latency. When AQE is enabled, stages are wrapped in QueryStages that are synchronously created. As a result, the stage interleaving patterns are consistent for running one query, making the query performance more stable and predictable.

spark.locality.wait=0s The default value of the parameter is 3s, which specifies the wait time for launching a data-local task before giving up and launching it on a less-local node. However, the waiting behavior can introduce instability in query performance due to the randomness of locality detection. As demonstrated in Figure 17(a) and Figure 17(b), the latency of a stage can vary significantly (changing from 11.7s to 18.4s) when different locality detection methods are employed. To ensure a stable query performance in our workload, we have fixed the `spark.locality.wait` parameter to 0s, thereby avoiding the waiting time for locality and achieving consistent and better query performance as shown in Figure 17(c). It is worth noting that in the production environment [28], the impact of locality is mitigated due to the high-speed network cards, which aligns with a near-zero waiting time.

spark.sql.adaptive.coalescePartitions.parallelismFirst=false We respect the recommendation on the Spark official website and set this parameter to false such that the advisory partition size will be respected when coalescing contiguous shuffle partitions.

spark.sql.adaptive.forceOptimizeSkewedJoin=false We follow the default setting for the parameter and avoid optimizing the skewed joins if it requires an extra shuffle. However, if needed, we can set it to true and adjust our tuning process to consider s_5 - s_7 as three additional plan-dependent parameters.

B.3 More Integration Evaluation

The framework of compile-time and runtime optimization is shown in Figure 18. We show per query latency comparison with a strong speed preference in TPC-H in Figure 19.

(a) Case 1: locality.wait=3s, costs 11.7s (b) Case 2: locality.wait=3s, cost 18.4s (c) Case 3: locality.wait=0s, cost 7.0s

Figure 17: The end-to-end stage latency comparison over different settings for spark.locality.wait.

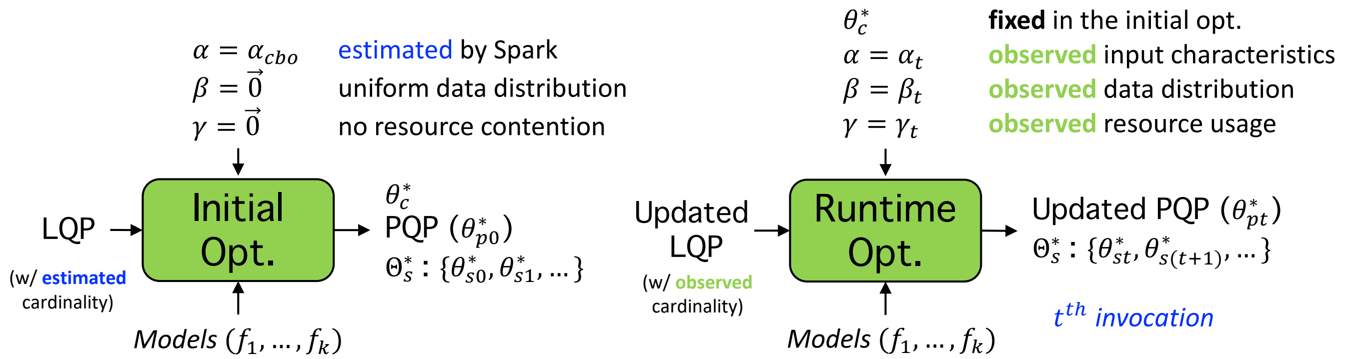


Figure 18: Compile-time Optimization and Runtime Optimization

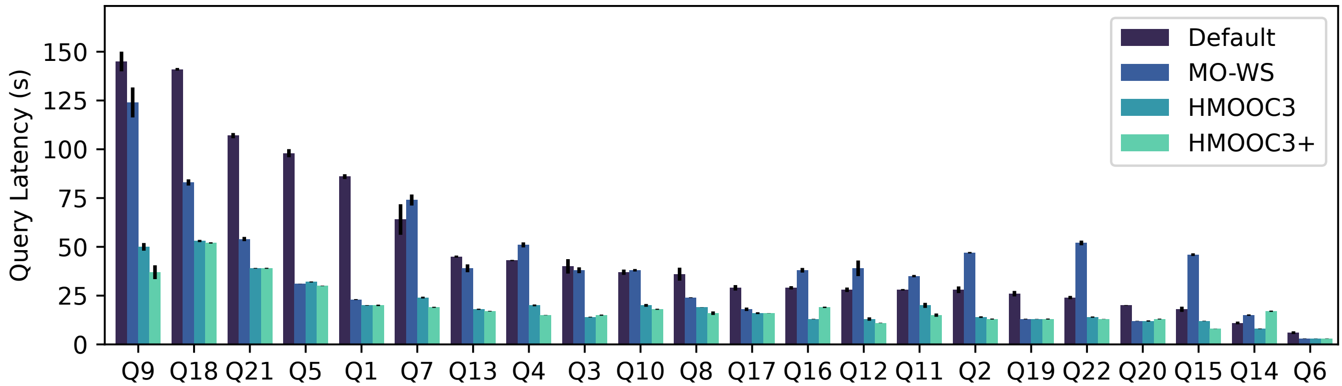


Figure 19: Per-query latency comparison with a strong speed preference in TPCH