Spark-based Cloud Data Analytics using Multi-Objective Optimization

Fei Song†, Khaled Zaouk†, Chenghao Lyu‡, Arnab Sinha†, Qi Fan†, Yanlei Diao†‡, Prashant Shenoy ‡
† Ecole Polytechnique; ‡ University of Massachusetts, Amherst
†{fei.song, khaled.zaouk, arnab.sinha, qi.fan, yanlei.diao}@polytechnique.edu; ‡{chenghao, shenoy}@cs.umass.edu

***Abstract*—Data analytics in the cloud has become an integral part of enterprise businesses. Big data analytics systems, however, still lack the ability to take task objectives such as user performance goals and budgetary constraints and automatically configure an analytic job to achieve these objectives. This paper presents UDAO, a Spark-based Unified Data Analytics Optimizer that can automatically determine a cluster configuration with a suitable number of cores as well as other system parameters that best meet the task objectives. At a core of our work is a principled *multi-objective optimization* (MOO) approach that computes a Pareto optimal set of configurations to reveal tradeoffs between different objectives, recommends a new Spark configuration that best explores such tradeoffs, and employs novel optimizations to enable such recommendations within a few seconds. Detailed experiments using benchmark workloads show that our MOO techniques provide a 2-50x speedup over existing MOO methods, while offering good coverage of the Pareto frontier. Compared to Ottertune, a state-of-the-art performance tuning system, UDAO recommends Spark configurations that yield 26%-49% reduction of running time of the TPCx-BB benchmark while adapting to different user preferences on multiple objectives.**

I. Introduction

As the volume of data generated by enterprises has continued to grow, big data analytics in the cloud has become commonplace for obtaining business insights from this voluminous data. Despite its wide adoption, current big data analytics systems such as Spark remain best effort in nature and typically lack the ability to take user objectives such as performance goals or cost constraints into account.

Determining an optimal hardware and software configuration for a big-data analytic task based on user-specified objectives is a complex task and one that is largely performed manually. Consider an enterprise user who wishes to run a mix of Spark analytic tasks in the cloud. First, she needs to choose the server hardware configuration from the set of available choices, e.g., from over 190 hardware configurations offered by Amazon's EC2 [1]. These configurations differ in the number of cores, RAM size, availability of solid state disks, etc. After determining the hardware configuration, the user also needs to determine the software configuration by choosing various runtime parameters. For the Spark platform, these runtime parameters include the *number of executors*, *cores per executor*, *memory per executor*, *parallelism* (for reduce-style transformations), *Rdd compression* (boolean), *Memory fraction* (of heap space), to name a few.

The choice of a configuration is further complicated by the need to optimize *multiple*, possibly conflicting, user objectives. Consider the following real-world use cases at data analytics companies and cloud providers (anonymized for confidentiality) that elaborate on these challenges and motivate our work:

Use Case 1 (Data-driven Business Users). A data-driven security company that runs thousands of cloud analytic tasks daily has two objectives: keep the *latency* low in order to quickly detect fraudulent behaviors and also reduce *cloud costs* that impose substantial operational expenses on the company. For cloud analytics, task latency can often be reduced by allocating more resources, but at the expense of higher cloud costs. Thus, the engineers face the challenge of deciding the cloud configuration and other runtime parameters that balance *latency* and *cost*.

Use Case 2 (Serverless Analytics). Cloud providers now offer databases and Spark for serverless computing [2], [23]. In this case, a database or Spark instance is turned off during idle periods, dynamically turned on when new queries arrive, and scaled up or down as the load changes over time. For example, a media company that uses a serverless offering to run a news site sees peak loads in the morning or as news stories break, and a lighter load at other times. The application specifies the minimum and maximum number of computing units (CUs) to service its workload across peak and off-peak periods; it prefers to minimize cost when the load is light and expects the cloud provider to dynamically scale CUs for the morning peak or breaking news. In this case, the cloud provider needs to balance between latency under different data loads and user cost, which directly depends on the number of CUs used.

Overall, choosing a configuration that balances multiple conflicting objectives is non-trivial—even expert engineers are often unable to choose between two cluster options for a single objective like latency [24], let alone choosing between dozens of cluster options for multiple competing objectives.

In this paper, we introduce a Spark-based Unified Data Analytics Optimizer (UDAO) that can automate the task of determining an optimal configuration for each Spark task based on multiple task objectives. Targeting cloud analytics, UDAO is designed with the following features:

First, UDAO aims to support a broad set of analytic tasks beyond SQL. Today, cloud analytics pipelines often mix SQL queries, ETL tasks based on SQL and UDFs, and machine learning (ML) tasks for deep analysis—this observation is revealed in our discussions with cloud providers, and is further supported by the recent development of the TPCx-BB benchmark [32] that mixes SQL, UDF, and ML tasks in the

same benchmark. UDAO unifies all of them in the general paradigm of dataflow programs and is implemented on top of Spark, a well-known unified analytics engine with both on-premise and serverless offerings in the cloud (e.g., [23]).

Second, UDAO takes as input an analytic task in the form of a dataflow program and a set of objectives, and produces as output a configuration with a suitable number of cores as well as other runtime parameters that best meet the task objectives. At the core of UDAO is a principled **multi-objective optimization** (MOO) approach that takes multiple, possibly conflicting, objectives, computes a Pareto-optimal set of configurations (i.e., not dominated by any other configuration in all objectives), and returns one from the set that best suits the objectives.

While MOO under numerical parameters has been studied in the optimization community [19], [8], we address the MOO problem specifically for designing a cloud analytics optimizer—our problem setting poses systems challenges such as sufficient coverage of the Pareto set under stringent time constraints. More specifically, these challenges include:

1. *Infinite parameter space*: Since Spark runtime parameters mix numerical and categorial parameters, there are potentially infinite configurations, only a small fraction of which belong to the Pareto set—most configurations are dominated by some Pareto optimal configuration for all objectives. Hence, we will need to efficiently search through an infinite parameter space to find those Pareto optimal configurations.

2. *Sufficient, consistent coverage of the Pareto Frontier*: The Pareto set over the multi-objective space is also called the *Pareto frontier*. Since we aim to use the Pareto frontier to recommend a system configuration that best explores tradeoffs between different objectives, the frontier should provide sufficient coverage of the objective space. As we shall show in the paper, existing MOO methods such as Weighted Sum [19] often fail to provide sufficient coverage. Further, the Pareto frontier must be consistent, i.e., the solutions computed with more CPU time should subsume the previous solutions. Randomized MOO solutions such as Evolutional methods [8] cannot guarantee consistent Pareto frontiers and hence can lead to contradicting recommendations, as we reveal in evaluation, which is highly undesirable for a cloud optimizer.

3. *Efficiency*: Most notably, configurations must be recommended under **stringent time requirements**, e.g., within a few seconds, to minimize the delay of starting a recurring task, or invoking or scaling a serverless task. Such systems constraints fundamentally distinguish our MOO work from the theoretical work in the optimization community [19], [8]. Such time constraints are aggravated when the optimizer needs to call complex performance models of the objectives, e.g., to understand the (estimated) latency of a particular configuration. Recent modeling work tends to use models of high complexity, e.g., Gaussian Processes or Deep Neural Networks [35], [38], and updates these models frequently as new training data becomes available. In this case, the MOO algorithm needs to recompute the Pareto frontier based on the updated model in order to recommend a new configuration that best suits the task objectives.

By way of designing our Spark-based optimizer, our paper makes the following contributions:

(1) We address the infinite search space issue by presenting a new approach for incrementally transforming a MOO problem to a set of constrained optimization (CO) problems, where each CO problem can be solved individually to return a Pareto optimal point.

(2) We address the coverage and efficiency challenges by designing practical Progressive Frontier (PF) algorithms to realize our approach. (i) Our first PF algorithm is designed to be *incremental*, i.e., gradually expanding the Pareto frontier as more computing time is invested, and *uncertainty-aware*, i.e., returning more points in regions of the frontier that lack sufficient information. (ii) To support complex learned models in recent work [7], [35], [38], we further develop an *approximate* PF algorithm that solves each CO problem efficiently for subdifferentiable models. (iii) We finally devise a *parallel*, approximate PF algorithm to further improve efficiency.

(3) We implement our algorithms into a Spark-based prototype. Evaluation results using batch and stream analytics benchmarks show that our approach produces a Pareto frontier within 2.5 seconds, and outperforms MOO methods including Weighted Sum [19], Normal Constraints [19], Evolutional methods [8], and multi-objective Bayesian optimization [10], [5] with 2-50x speedup, while offering better coverage over the frontier and enabling exploration of tradeoffs such as cost-latency or latency-throughput. When compared to Ottertune [35], a state-of-the-art performance tuning system, our approach recommends configurations that yield 26%-49% reduction of total running time of the TPCx-BB benchmark [32] while adapting to different application preferences on multiple objectives and accommodating a broader set of models.

II. Background and System Design

In this section, we discuss requirements and constraints from real-world use cases that motivate our system design.

A. Use Cases and Requirements

We model an analytic task as a dataflow program (a directed graph of data collections flowing between operations), which is used as the programming model in big data systems such as Spark [37]. We assume that each Spark task has user- or provider-specified objectives, referred to as *task objectives*, that need to be optimized during execution. To execute the task, the system needs to determine a *cluster execution plan* with runtime parameters instantiated. As stated before, these parameters control resource allocation (num. of cores, num. of executors, memory per executor), the degree of parallelism (for reduce-style transformations), granularity of scheduling, compression options, shuffling strategies, etc. An executing task using this plan is referred to as a *job* and the runtime parameters are collectively referred as the *job configuration*.

The goal of our Spark-based optimizer is: *given a data flow program and a set of objectives, compute a job configuration*

that optimizes these objectives and adapt the configuration quickly if either the load or task objectives change.

To meet real world analytics needs, we present two concrete use cases with their requirements from our discussions with analytic and cloud companies, and design our optimizer (UDAO) accordingly to support these use cases.

1. Recurring user workloads. It is common for analytical users to issue repeated jobs in the form of daily or hourly batch jobs [38] . Sometimes stream jobs can be repeated as well: under the lambda architecture, the batch layer runs to provide perfectly accurate analytical results, while the speed layer offers fast approximate analysis over live streams; the results of these two layers are combined to serve a model. As old data is periodically rolled into the batch job, the streaming job is restarted over new data with a clean state.

UDAO is designed to work for recurring workloads: for each scheduled job of a recurring task, once a user request is sent to UDAO, it recommends a configuration under a few seconds to improve job performance towards user-specified objectives.

2. Serverless analytics. As noted before, serverless analytics using databases (DBs) or Spark are becoming common in cloud computing. Each invocation of a serveless task by the cloud platform requires a configuration that meets multiple objectives—to provide low query latency to end-users while using the least computing units (CUs) for the expected load. Furthermore, auto scaling features imply that new configurations need to be computed quickly to react to load changes.

UDAO is designed to also support serverless workloads: whenever the cloud platform needs to launch a serverless analytic task or scale it to adapt to the load, the cloud platform sends a request to UDAO and within a few seconds receives a recommended configuration balancing latency and cost.

B. System Overview

To meet stringent time constraints, we begin our system design by separating model learning and MOO into two asynchronous procedures: The time-consuming modeling process is performed offline by a model server whenever new training data becomes available. MOO runs online on-demand and uses the most recent model to compute a new configuration, with the delay of a few seconds. This approach distinguishes UDAO from DBMS performance tuning systems [35], [39] that couple modeling and optimization steps in an online iterative tuning session, which takes 15-45 mins to run for each query workload. The implications of this change are twofold: (i) It enables MOO to work with a range of models. More specifically, UDAO's MOO algorithm works with complex learned models represented by Gaussian Processes (GPs) or Deep Neural Networks (DNNs), besides simple closed-form regression functions, whereas the optimization methods in [35], [39] work only for the specific model (GP or DNN) of choice. (ii) The model server can train a new model in the background as new training data becomes available. When MOO needs to be run for a given task, the model for this task may have been updated. Hence, the speed to compute a Pareto frontier based on the new model is a key performance goal.

Figure 1(a) shows the design of UDAO based on this architectural change, with Spark as the underlying analytics engine. It takes as input a sequence of user or provider initiated requests, where each request specifies an analytic task and a set of objectives, $(F_1, \ldots, F_k)$. Currently, UDAO offers a set of objectives for external requests to choose from, including: 1) average latency, for both batch and stream processing; 2) throughput for stream processing; 3) CPU utilization; 4) IO load; 5) network load; 6) resource cost in CPU cores; 7) resource cost in CPU-hour (latency x CPU cores); and 8) resource cost as a weighted combination of CPU-hour and IO cost (inspired by Severless DBs [2]), while more options can be added in the future as application needs arise. Besides the chosen objectives, requests can optionally specify value constraints on these objectives, $F_i \in [F_i^L, F_i^U]$, as well as preferences on these objectives as a weight vector $(w_1, \ldots, w_k)$, $0 \leq w_i \leq 1$ and $\sum_{i=1}^{k} w_i = 1$.

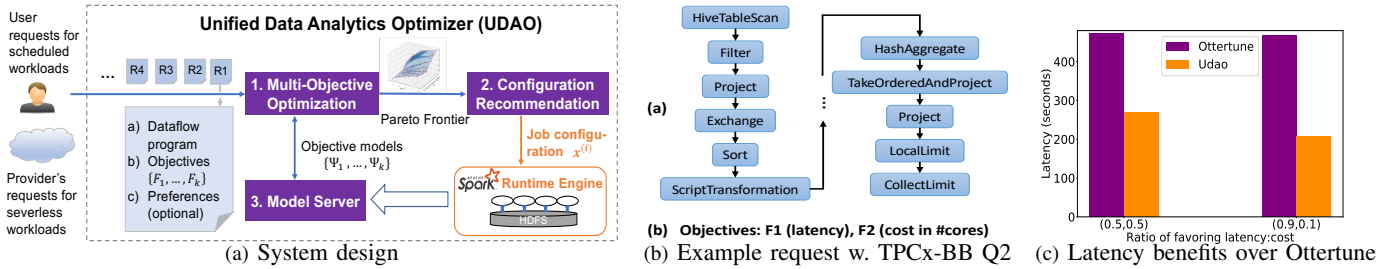
Figure 1(b) shows an example request, including the dataflow program of Q2 from the TPCx-BB [32] benchmark, which mixes SQL with UDFs, and the application choice of latency and resource cost (CPU cores) as objectives.

UDAO handles each request as follows: If a Spark task runs for the first time or is specified with new objectives, no predictive models are available for these objectives yet. Its job will be to run with a default configuration $\boldsymbol{x}^1$. Our optimizer assumes that a separate model server will collect traces during job execution, including (i) system-level metrics, e.g., time measurements of different steps, bytes read and written, and fetch wait time from the Spark engine; (ii) observed values of task objectives such as latency and cost. The model server uses these traces offline asynchronously to compute task-specific models $(\Psi_1, \ldots, \Psi_k)$, one for each chosen objective.

When the analytic task runs the next time, the multi objective optimization (MOO) module will contact the model server and retrieve the task-specific predictive models. Then it searches through the space of configurations and computes a set of Pareto-optimal configurations. Based on insights revealed in the Pareto frontier, the recommendation module chooses a new configuration, $\boldsymbol{x}^2$, that meets all user constraints and best explores the tradeoffs among different objectives. Future runs of the task will use this new configuration.

If the application decides to adjust the constraints on the objective (e.g., the service provider specifies a new bound $[F_i^L, F_i^U]$ to increase the throughput requirement when higher data rates occur) or adjust the weight factor (e.g., favoring latency more to cost in the coming hour), the optimizer can quickly return a new configuration from the computed Pareto frontier. As the model server continues to collect additional samples from recurring executions of this task as well as others, it may periodically update the task's predictive models. Upon the next run of the task, if updated models become available, the Pareto frontier will be recomputed by re-running the MOO and a new configuration will be recommended.

For the example task in Figure 1(b), Figure 1(c) shows the effect of optimization using UDAO against OtterTune [35], where the application first specified (0.5, 0.5) weights for



(a) System design (b) Example request w. TPCx-BB Q2 (c) Latency benefits over Ottertune

Fig. 1. Overview of UDAO, an unified data analytics optimizer built on top of Spark

latency and cost and then later (0.9, 0.1) weights to favor latency to cost. UDAO can achieve 43%-56% reduction in latency compared to OtterTune while adapting to user preferences (which will be detailed in our performance study).

Remarks on modeling choices. Since our focus in this paper is on MOO rather than modeling, we briefly discuss relevant modeling techniques here. As UDAO is designed for Spark workloads, various modeling options can be used to provide such models: (1) *Handcrafted models:* Domain knowledge and workload profiling were used to develop specific regression models for the Spark platform [36], where different hardware profiles can be collected to customize these models. Such models employ relatively simple function shapes (linear or low-degree polynomial) on a small set of resource parameters (e.g., the number of nodes). (2) *Learned models:* Recent techniques can automatically learn predictive models, including function shapes, coefficients, etc. from runtime traces [7], [35], [38], [39]. These models tend to employ complex functions built on larger numbers of parameters. Section V discusses how UDAO incorporates these models using the model server.

III. Progressive Frontier Approach

We now present our Progressive Frontier framework for solving the MOO problem. Since our focus is on systems aspects of MOO-driven cloud analytics, we present key concepts in brief and refer the reader to [29] for details and proofs.

Problem III.1. Multi-Objective Optimization (MOO).

$$\arg\min_{\boldsymbol{x}} \quad f(\boldsymbol{x}) = \begin{bmatrix} F_1(\boldsymbol{x}) = \Psi_1(\boldsymbol{x}) \\ \dots \\ F_k(\boldsymbol{x}) = \Psi_k(\boldsymbol{x}) \end{bmatrix} \tag{1}$$

$$s.t. \quad \boldsymbol{x} \in \Sigma \subseteq \mathbb{R}^d$$
$$F_i^L \le F_i(\boldsymbol{x}) \le F_i^U, \quad i = 1, ..., k$$

where $\boldsymbol{x}$ is the job configuration with d parameters, $F_i(\boldsymbol{x})$ generally denotes each of the k objectives as a function of $\boldsymbol{x}$, and $\Psi_i(\boldsymbol{x})$ refers particularly to the predictive model derived for this objective. If an objective (e.g., throughput) favors larger values, we add the minus sign to the objective function to transform it to a minimization problem.

In general, multi-objective optimization (MOO) leads to a set of solutions rather than a single optimal solution.

Definition III.1. Pareto Optimality: In the objective space $\Phi \in \mathbb{R}^k$, a point $\boldsymbol{f'}$ Pareto-dominates another point $\boldsymbol{f''}$ iff $\forall i \in [1,k], f_i' \le f_i''$ and $\exists j \in [1,k], f_j' < f_j''$. A point $\boldsymbol{f^*}$ is **Pareto Optimal** iff there does not exist another point $\boldsymbol{f'}$ that Pareto-dominates it. For an analytic task, the **Pareto Set (Frontier)** $\mathcal{F}$ includes all the Pareto optimal points in the objective space Φ, and is the solution to the MOO problem.

We further require the Pareto frontier to be computed with *good coverage*, *high efficiency*, and *consistency*. Among existing MOO methods, Weighted Sum (WS) [19] is known to have *poor coverage* of the Pareto frontier [20]. Normalized Constraints (NC) [21] suffers from *efficiency* issues: it uses a pre-set parameter k to indicate the number of Pareto points desired but often returns fewer points than k; if more points are needed to cover the Pareto frontier, a large value k' will be tried, by starting the computation from scratch. Evolutionary Methods (Evo) [8] are randomized methods to approximately compute a frontier set, which suffer from the problem of *inconsistency*: The Pareto frontier built with k' points can be inconsistent with that built with k points, as our evaluation results show. Multi-objective Bayesian Optimization (MOBO) extends the Bayesian approach to modeling an unknown function with an acquisition function for choosing the next point(s) to explore so that these new points are likely to be Pareto points. It suffers from *inefficiency*, taking a long time to return a decent Pareto set, as our evaluation results show.

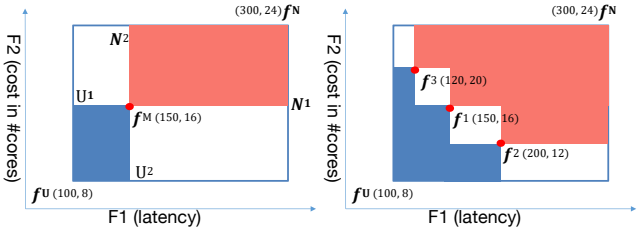
To meet all the above requirements, we introduce a new approach, called **Progressive Frontier**. It incrementally transforms MOO to a series of constrained single-objective optimization problems, which can be solved individually.

Definition III.2. Uncertain Space: A Pareto optimal point is denoted as a reference point $r^i \in \Phi$ if it achieves the minimum for objective F_i $(i = 1, ..., k)$. A point $\boldsymbol{f^U} \in \Phi$ is a **Utopia** point iff for each $j = 1, ..., k$, $f_j^U = \min_{i=1}^k \{r_j^i\}$. A point $\boldsymbol{f^N} \in \Phi$ is a **Nadir** point iff for each $j = 1, ..., k$, $f_j^N = \max_{i=1}^k \{r_j^i\}$. Given a hyperrectangle formed by the Utopia Point $\boldsymbol{f^U}$ and Nadir Point $\boldsymbol{f^N}$ in the objective space, the **Uncertain Space** is defined as the volume of this hyperrectangle.

Next we describe a method to find *one* Pareto point by solving a single-objective constrained optimization problem.

Definition III.3. Middle Point Probe: Given a hyperrectangle formed by $\boldsymbol{f^U} = (f_1^U, \dots, f_k^U)$ and $\boldsymbol{f^N} = (f_1^N, \dots, f_k^N)$, the middle point $\boldsymbol{f^M}$ bounded by $\boldsymbol{f^U}$ and $\boldsymbol{f^N}$ is defined as the solution to **Constrained Optimization (CO)** of:

$$\boldsymbol{x}_{\mathcal{C}} = \arg\min_{\boldsymbol{x}} \quad F_i(\boldsymbol{x}),$$
$$\text{subject to} \quad f_j^U \le F_j(\boldsymbol{x}) \le \frac{(f_j^U + f_j^N)}{2}, \quad j \in [1,k]. \tag{2}$$



(a) Middle point probe. (b) multiple Pareto points.

Fig. 2. Uncertain space in 2D (latency, cost) space for TPCx-BB Q2. $\boldsymbol{f^U}$ and $\boldsymbol{f^N}$ are the Utopia and Nadir points, respectively. In (a), $\boldsymbol{f^M}$ is the solution to the middle point probe. In (b), ($\boldsymbol{f^1}$, $\boldsymbol{f^2}$, $\boldsymbol{f^3}$) represent the solutions of a series of middle point probes.

where we can choose any i to be the objective to minimize.

Fig. 2(a) illustrates a middle point probe for our running example, TPCx-BB Q2 in Fig. 1(b). Here, the 2D space is defined over F_1 (latency) and F_2 (cost in #cores), the Utopia point $\boldsymbol{f^U}= (100, 8)$ denotes the hypothetical best performance (low latency using 8 cores), and the Nadir point $\boldsymbol{f^N}= (300, 24)$ denotes the worst (high latency using 24 cores). The middle point probe generates a CO problem, $\mathcal{C}_{F_1F_2}$: $\min F_1$ (lat.) such that $F_1 \in [100, 200]$ and F_2 (cost) in [8,16], and returns a Pareto point, $\boldsymbol{f^M} = (150, 16)$.

After finding the middle point $\boldsymbol{f^M}$, the sub-hyperrectangle enclosed by $\boldsymbol{f^M}$ and $\boldsymbol{f^N}$, shaded in red, contains only points dominated by $\boldsymbol{f^M}$; hence no Pareto points can exist there. The sub-hyperrectangle enclosed by $\boldsymbol{f^U}$ and $\boldsymbol{f^M}$, shaded in blue, must be empty; otherwise $\boldsymbol{f^M}$ cannot be Pareto optimal. This means that we can safely discard these two colored sub-hyperrectangles. The uncertain space is then reduced to two unshaded sub-hyperrectangles, enclosed by (U^1, N^1) and (U^2, N^2), respectively, which can be added to a queue. If we continue to take each sub-hyperrectangle from the queue and apply the Middle Point Probe iteratively, we can further reduce the uncertain space as shown in the unshaded region in Figure 2(b). Such an iterative procedure is called **Iterative Middle Point Probes**. This procedure can be extended naturally to hyperrectangles in k-dimensional objective space.

In general, we have the following result on the returned solution set of the Iterative Middle Point Probes procedure.

Proposition III.1. If we start the Iterative Middle Point Probes procedure from the initial Utopia and Nadir points, and let it terminate until the uncertain space becomes empty, then in the 2D case, our procedure guarantees to find all the Pareto points if they are finite. In high-dimensional cases, it is guaranteed to find a subset of Pareto optimal points.

IV. Practical PF Algorithms

In this section, we present practical algorithms to implement our Progressive Frontier (PF) approach. Note that most MOO algorithms suffer from exponential complexity in the number of the objectives, k. This is because the number of non-dominated points tends to grow quickly with k and the time complexity of computing the volume of dominated space grows super-polynomially with k [8]. For this reason, the MOO literature refers to optimization with up to 3 objectives as **multi-objective** optimization, whereas optimization with more than 3 objectives is called **many-objective** optimization and handled using different methods such as preference modeling [8] or fairness among different objectives [31].

Since we aim to develop a practical cloud optimizer, most of our use cases fall in the scope of multi-objective optimization. However, a key systems challenge is to compute the Pareto frontier in a few seconds, which hasn't been considered previously [8]. To achieve our performance goal, we present a suite of techniques, including *uncertainty-aware incremental computation*, *fast approximation*, and *parallel computation*.

A. A Deterministic Sequential Algorithm

We first present a deterministic, sequential algorithm that implements the Progressive Frontier (PF) approach, referred to as PF-S. This algorithm has two key features:

(1) *Incremental*: It first constructs a Pareto frontier $\tilde{\mathcal{F}}$ with a small number of points and then expands $\tilde{\mathcal{F}}$ with more points as more time is invested. This feature is crucial because finding one Pareto point is already expensive due to being a mixed-integer nonlinear programming problem [11]. Hence, one cannot expect the optimizer to find all Pareto points at once. Instead, it produces n_1 points first (e.g., those that can be computed within the first second), and then expands with additional n_2 points, afterwards n_3 points, and so on.

(2) *Uncertainty-aware*: The algorithm returns more points in the regions of the Pareto frontier that lack sufficient information. If the Pareto frontier includes many points, under a time constraint we can return only a subset of them. Further, in high-dimensional objective space, our PF approach can guarantee to find only a subset of Pareto points. In both cases, the uncertainty aware property means that the subset returned is likely to capture the major trend on the Pareto frontier.

To do so, we augment the Iterative Middle Point Probes method (§III) by further deciding how to choose the **best** sub-hyperrectangle to probe next. We do so by defining a measure, the *volume of uncertain space*, to capture how much the current frontier $\tilde{\mathcal{F}}$ may deviate from the true yet unknown frontier $\mathcal{F}$. This measure can be calculated from a related set of sub-hyperrectanges, which allows us to rank the sub-hyperrectangles that have not been probed. Among those, the sub-hyperrectangle with the largest volume will be chosen to probe next, thus reducing the uncertain space as fast as we can. Algorithm 1 shows the details, where the main steps are:

Init: Find the reference points by solving k single-objective optimization problems. Form the initial Utopia and Nadir (U^0 and N^0) points, and construct the first hyperrectangle. Prepare a priority queue in decreasing order of hyperrectangle volume, initialized with the first hyperrectangle.

Iterate: Pop a hyperrectangle from the priority queue. Apply the middle point probe to find a Pareto point, $\boldsymbol{f^M}$, in the current hyperrectangle, which is formed by U^i and N^i and has the **largest** volume among all the existing hyperrectangles. Divide the current hyperrectangle into 2^k sub-hyperrectangles, discard those that are dominated by $\boldsymbol{f^M}$, and calculate the volume of the others. Put them in to the priority queue.

Terminate: when we reach the desired number of solutions.

Algorithm 1 Progressive Frontier-Sequential (PF-S)

Require: k lower bounds(LOWER): $lower^j$,
k upper bounds(UPPER): $upper^j$,
number of points: M
1: $PQ \leftarrow \phi$ {PQ is a priority queue sorted by hyperrectangle volume}
2: $plan_i \leftarrow optimize^i(LOWER, UPPER)$ {Single Objective Optimizer takes LOWER and UPPER as constraints and optimizes on ith objective}
3: $Utopia, Nadir \leftarrow computeBounds(plan_1, \ldots, plan_k)$
4: $volume \leftarrow$ computeVolume$(Utopia, Nadir)$
5: $seg \leftarrow (Utopia, Nadir, volume)$
6: PQ.put(seg)
7: $count \leftarrow k$
8: **repeat**
9: $seg \leftarrow PQ$.pop()
10: $Utopia \leftarrow seg.Utopia$; $Nadir \leftarrow seg.Nadir$
11: $Middle \leftarrow (Utopia + Nadir)/2$
12: $Middle_i \leftarrow optimize^i(Utopia, Middle)$ {Constraint Optimization on i-th objective}
13: $\{plan\} \leftarrow Middle_i$
14: $count+ = 1$
15: $\{rectangle\} = generateSubRectangles(Utopia, Middle, Nadir)$ {return 2^k - 1 rectangles, represented by each own Utopia and Nadir}
16: **for** each rectangle in {rectangle} **do**
17: $Utopia \leftarrow rectangle.Utopia$; $Nadir \leftarrow rectangle.Nadir$
18: $volume \leftarrow$ computeVolume$(Utopia, Nadir)$
19: $seg \leftarrow (Utopia, Nadir, volume)$
20: PQ.put(seg)
21: **end for**
22: **until** $count > M$
23: $output \leftarrow filter(\{plan\})$ {filter dominated points}

Filter: Check the result set, and remove any point dominated by another one in the result set.

B. Multi-Objective Gradient Descent

We next consider the subroutine, $optimize()$, that solves each constrained optimization problem (line 13 of Algorithm 1). As our objective functions are given by the models, $\Psi_i(\boldsymbol{x})$, $i = 1 \ldots k$, returned from the model server, they are often non-linear and the variables in $\boldsymbol{x}$ can be integers and real numbers. This problem reduces to mixed-integer nonlinear programming (MINLP) and is NP-hard [11]. There are no general solvers that work for every MINLP problem [22]. Most of the MINLP solvers [22] assume certain properties of the objective function, e.g., twice continuously differentiable (Bonmin [3]) or factorable into the sumproduct of univariate functions (Couenne [4]), which do not suit learned models such as Deep Neural Networks (DNNs). The most general MINLP solver, Knitro [14], supports complex models but runs very slowly, e.g., 42 (17) minutes for solving a *single* one-objective optimization problem based on a DNN (GP) model.

Therefore, we develop a new solver that uses a customized gradient descent (GD) approach to approximately solve constrained optimization (CO) problems for MOO (see Fig 3(a)).

In the first step, we transform variables to prepare for optimization by following the common practice in machine learning: Let $\mathbf{x}$ be the original set of parameters, which can be categorical, integer, or continuous variables. If a variable is categorical, we use one-hot encoding to create dummy variables. For example, if x_d takes values {a, b, c}, we create three boolean variables, x_d^a, x_d^b, and x_d^c, among which only one takes the value '1'. Afterwards all the variables are normalized to the range [0, 1], and boolean variables and (normalized) integer variables are relaxed to continuous variables in [0, 1]. As such, the CO problem deals only with continuous variables in [0,1], which we denote as $\boldsymbol{x} = x_1, \ldots, x_D \in [0, 1]$. After a solution is returned for the CO problem, we set the value for a categorical attribute based on the dummy variable with the highest value, and round the solution returned for a normalized integer variable to its closest integer.

Next, we focus on the CO problem. Our design of a **Multi-Objective Gradient Descent (MOGD) solver** uses carefully-crafted loss functions to guide gradient descent to find the minimum of a target objective while satisfying a variety of constraints, where both the target objective and constraints can be specified on complex models, e.g., DNNs and GPs.

1. Single objective optimization. As a base case, we consider single-objective optimization, *minimize* $F_1(\boldsymbol{x}) = \Psi_1(\boldsymbol{x})$. For optimization, we set the loss function simply as, $\mathcal{L}(\boldsymbol{x}) = F_1(\boldsymbol{x})$. Then starting from an initial configuration, $\boldsymbol{x}^0$, gradient descent (GD) will iteratively adjust the configuration to a sequence $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n$ in order to minimize the loss, until it converges to a local minimum or reaches a maximum of steps.

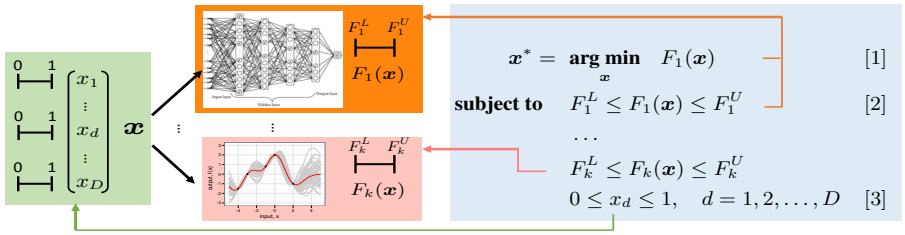
To increase the chance of hitting a better local minimum (closer to the global minimum), we use a *multi-start* method to try gradient descent from multiple initial points, and finally choose $\boldsymbol{x}^*$ that gives the smallest value among these trials. Further, among GD variants (e.g, momentum, SGD) we use Adaptive Moment Estimation (Adam) [25], recommended as an overall best choice for achieving better local minima [25]. To cope with the constraint, $0 \leq x_d \leq 1$, we restrict GD such that when it tries to push x_d out of the range, we set x_d to the boundary value. In future iterations, it can still adjust other variables, but not x_d across its boundaries, to reduce the loss.

2. Constrained optimization. Next we consider a constrained optimization (CO) problem, as shown in Figure 3(a). WLOG, we treat F_1 as the target objective, and $F_j \in [F_j^L, F_j^U]$, $j = 1, \ldots, k$, as constraints, which are set by our middle point probe method (Eq. 2). To solve the CO problem, we seek to design a new loss function, $\mathcal{L}(\boldsymbol{x})$, such that by minimizing this loss, we can minimize $F_1(\boldsymbol{x})$ while at the same time satisfying all the constraints. Our proposed loss function is as follows:

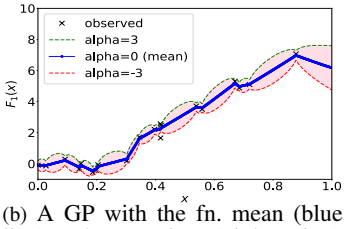
$$\begin{aligned}\mathcal{L}(\boldsymbol{x}) = \quad & \mathbb{1}\{0 \leq \hat{F}_1(\boldsymbol{x}) \leq 1\} \cdot \hat{F}_1(\boldsymbol{x})^2 + \\ & \sum_{j=1}^{k} \mathbb{1}\{\hat{F}_j(\boldsymbol{x}) > 1 \vee \hat{F}_j(\boldsymbol{x}) < 0\}[(\hat{F}_j(\boldsymbol{x}) - \frac{1}{2})^2 + P]\end{aligned} \tag{3}$$

where $\hat{F}_j(\boldsymbol{x}) = \frac{F_j(\boldsymbol{x}) - F_j^L}{F_j^U - F_j^L}$, $j \in [1, k]$, denotes the normalized value of each objective. Since the range of each objective function $F_j(\boldsymbol{x})$ varies, we normalize each objective according to its upper and lower bounds, so that a valid objective value $\hat{F}_j(\boldsymbol{x}) \in [0, 1]$. Further, P is a constant for extra penalty.

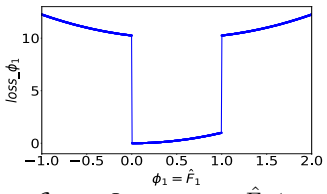
Fig. 3(c)-3(f) illustrate the loss function for the CO problem, $\mathcal{C}_{F_1 F_2}$: $\min F_1$ (lat.) such that $F_1 \in [100, 200]$ and F_2 (cost) in [8,16], shown in the previous section for TPCx-BB Q2.



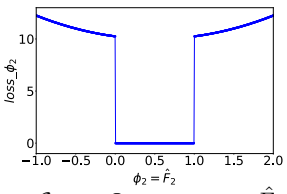
(a) Constrained optimization with k objectives (F_1 is modeled by a DNN and F_k by a GP)



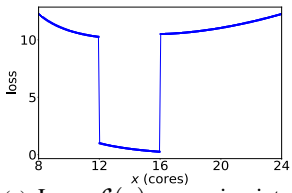
(b) A GP with the fn. mean (blue line) and uncertainty (pink region)



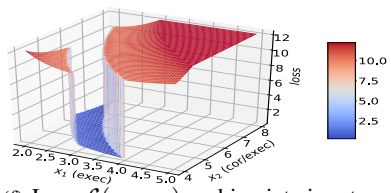
(c) $\mathcal{C}_{F1F2}$: Loss term on $\hat{F}_1$ (normalized lat.) minimizes it while pushing it into [0,1] (normalized range for $F_1 \in$ [100,200])



(d) $\mathcal{C}_{F_1F_2}$: Loss term on $\hat{F}_2$ (normalized cost) pushes it into [0,1] (normalized range for cost $\in$ [8,16])



(e) Loss $\mathcal{L}(x)$ on univariate input x (#cores), where F_1 (lat.) $= max(100, \frac{2400}{x})$, and F_2 (cost) $= min(24, x)$



(f) Loss $\mathcal{L}(x_1, x_2)$ on bivariate input x_1 (#exec) and x_2 (#cores/exec), where F_1 (lat.) $= max(100, \frac{2400}{min(24, x_1 x_2)})$, and F_2 (cost) $= min(24, x_1 x_2)$

Fig. 3. Constrained optimization with k objectives, and an example $\mathcal{C}_{F_1F_2}$ "min F_1 (lat.) such that $F_1 \in [100, 200]$ and F_2 (cost) in [8,16]"

Fig. 3(c)-3(d) illustrate the breakdown of the loss terms for F_1 and F_2. The loss for F_1 has two terms. When F_1 falls in the constraint region [100,200] (or in the normalized form, $0 \le \hat{F}_1 \le 1$), the first term of the loss penalizes a large value of F_1, and hence minimizing the loss helps reduce F_1. The second term of the loss aims to push the objective into its constraint region. If F_1 cannot meet the constraints ($\hat{F}_j > 0 \vee \hat{F}_j < 1$), it contributes a loss according to its distance from the valid region. The extra penalty, P, ensures that the loss for F_1 if it falls outside its valid region is much higher than that if F_1 lies in the region. Fig. 3(c) shows the effect of these two terms on F_1. In comparison, the loss for F_2 has only the second term, pushing it to meet its constraint, as shown in Fig. 3(d).

Fig. 3(e)-3(f) further illustrate the loss, $\mathcal{L}(\boldsymbol{x})$, given that $F_1(\boldsymbol{x})$ and $F_2(\boldsymbol{x})$ are both functions of system parameters $\boldsymbol{x}$. For simplicity, Fig. 3(e) shows the loss, $\mathcal{L}$, over univariate input x (#cores), assuming two simple models for $F_1(x)$ and $F_2(x)$. In practice, #cores is not a system parameter, but defined over two other parameters x_1 (#executors) and x_2 (#cores per executor). Fig. 3(f) shows the loss over x_1 and x_2. Given complex models on multiple parameters, the surface of $\mathcal{L}(\boldsymbol{x})$ quickly becomes complex. Nevertheless, the loss will guide GD such that by minimizing $\mathcal{L}$, it is likely to find the $\boldsymbol{x}$ value that is an approximate solution to the CO problem.

Note that GD usually assumes the loss function $\mathcal{L}$ to be differentiable, but our loss function is not at specific points. However, we only require $\mathcal{L}$ to be **subdifferentiable**: for a point $\boldsymbol{x}$ that is not differentiable, we can choose a value between its left derivative and right derivative, called a subgradient. Machine learning libraries allow subgradients to be defined by the user program and can automatically handle common cases including our loss functions for DNNs and GPs.

3. Handling model uncertainty. We have several extensions of our MOGD solver. Most notably, we extend to support *model uncertainty*: when our objective functions use learned models, these models may not be accurate before sufficient training data is available. Hence, our optimization procedure takes into account model uncertainty when recommending an optimal solution. We leverage recent machine learning methods that support a regression task, $F(\boldsymbol{x})$, with both expected value $\mathbb{E}[F(\boldsymbol{x})]$ and variance, $\texttt{Var}[F(\boldsymbol{x})]$; such methods include Gaussian Processes [27], with an example shown in Figure 3(b), and Bayesian approximation for DNNs [9]. Given $\mathbb{E}[F(\boldsymbol{x})]$ and $\texttt{Var}[F(\boldsymbol{x})]$, we only need to replace each objective function, $F_j(\boldsymbol{x})$, with $\tilde{F}_j(\boldsymbol{x}) = \mathbb{E}[F_j(\boldsymbol{x})] + \alpha \cdot \texttt{std}[F_j(\boldsymbol{x})]$, where α is a small positive constant. Here, $\tilde{F}_j(\boldsymbol{x})$ offers a more conservative estimate of $F_j(\boldsymbol{x})$ for solving a minimization problem, given the model uncertainty. Then we use $\tilde{F}_i(\boldsymbol{x})$ to build the loss function in Eq. 3 to solve the CO problem.

C. Approximate and Parallel PF Algorithms

Approximate Sequential (PF-AS): The PF Approximate Sequential algorithm (PF-AS) implements SO optimization (Line 2 of Algorithm 1) and constrained optimization (Line 13) using our MOGD solver. It yields an approximate Pareto set since the solution of each CO problem can be suboptimal. In fact, the SOTA MINLP solver, Knitro [14], also returns approximate solutions due to complex, non-convex properties of our objective functions, despite long running time.

Approximate Parallel (PF-AP): We further propose a parallel algorithm (PF-AP) that differs from the approximate sequential algorithm in that to probe a given hyperrecetange, we partition it into a l^k grid and for each grid cell, construct a CO problem using the the Middle Point Probe (Eq. 2). We send these l^k CO problems to our MOGD solver simultaneously. Internally, our solver will solve these problems in parallel (using multi-threading). Some of the cells will not return any Pareto point and hence will be discarded. For each of those cells that returns a Pareto point, the Pareto point breaks the cell into a set of sub-hyperrectangles that will be added to

a priority queue for probing later. Inside the queue, the sub-hyperrectangle with the largest volume is removed, as before. We then further partition it into l^k cells and ask the solver to solve their corresponding CO problems simultaneously. This process terminates when the queue becomes empty.

V. UDAO Implementation

UDAO is built on top of Spark with three key modules:

Model Server. UDAO's model server can take handcrafted subdifferentiable regression functions (e.g., [36]) to model task objectives. In addition, it supports two automatic tools to learn models from runtime traces: (i) GP models from OtterTune [35]: we chose OtterTune [35] over other tools [7], [39] because it outperforms iTuned [7], another GP-based tool, due to the ability to map a new query against all past queries in model building, while CDBTune [39] cannot return a regression function explicitly for *each* objective as required by our system. (2) Our custom DNN models [38] can further extract workload encodings for blackbox programs using advanced autoencoders [38] to improve prediction.

For the Spark platform, our model server takes a set of $\sim$40 parameters and a list of objectives such as latency, throughput, IO load, cost in CPU cores, and cost in CPU-hour (the full list was given in §II-B). The key steps in modeling include:

1) Training Data Collection: We distinguish online workloads, whose runs are invoked only by the user, from offline workloads, which the model server can sample intensively, e.g., an existing benchmark or some workloads offered by the user for sampling. We sample 100's configurations for each offline workload using (a) *heuristic sampling* based on Spark best practices and (b) *Bayesian optimization* [26] for exploring configurations that are likely to minimize latency. In contrast, we create a small sample of configurations (of size 6 to 30) for each online workload to reflect the constraint that they usually do not have many configurations. Our training set includes runtime traces from both offline and online workloads.

2) Feature Engineering: We construct features from runtime traces by following standard ML steps: filtering features with a constant value; normalizing numerical features; one-hot encoding for categorical variables; and knob selection, for which we follow OtterTune's practice to select $\sim$10 most important knobs (parameters) by mixing results from a LASSO-based selection method and Spark recommendations.

3) Model Training: As we collect training data over months, we train the model periodically and checkpoint the best model weights. Inspired by industry practice [28], when receiving a large trace update (e.g., 5000 new traces), we retrain the model via hyper-parameter tuning; with a small trace update (e.g., 1000 new traces), we train incrementally by fine-tuning the model from the latest checkpoint. After training on our full TPCx-BB dataset, the largest DNN model has 4 hidden layers, each with 128 nodes and ReLU as activation function, with backpropogation ran by Adam [25]. We observe the running time of retraining (incremental training) to be up to 6 hours using 5 servers (20 min. using one server), which are all background processes and do not affect MOO for online jobs.

Our model sever is implemented using PyTorch. The trained models interface with MOO through network sockets.

MOO. UDAO's MOO module is implemented in Java and invokes a solver for constrained optimization (CO). Our system supports several solvers including our MO-GD solver (§IV-B) and the Knitro [14] solver. To solve a single CO problem, Knitro with 16 threads takes 17 and 42 minutes to run on GP and DNN models, respectively. In contrast, MOGD with 16 threads takes 0.1-0.5 second while achieving the same or lower value of the target objective. Therefore, we use MOGD as the default solver in our sysyem.

Recommendation. Once a Pareto set is computed for a workload, our optimizer employs a range of strategies to recommend a new configuration from the set. We highlight the most effective strategies below and describe other recommendation strategies in our technical report [29].

First, the Utopia Nearest (UN) strategy chooses the Pareto point closest to the Utopia point $\boldsymbol{f}^{\boldsymbol{U}}$, by computing the Euclidean distance of each point in the Pareto set $\tilde{\mathcal{F}}$ to $\boldsymbol{f}^{\boldsymbol{U}}$ and returning the point that minimizes the distance.

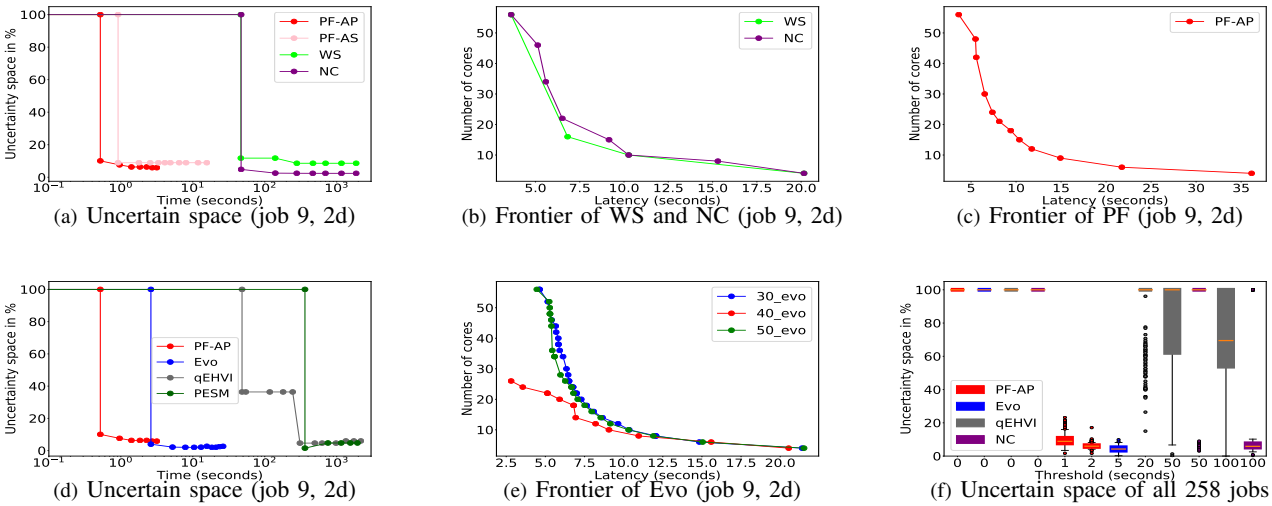
A variant is the Weighted Utopia Nearest (WUN) strategy, which uses a weight vector, $\boldsymbol{w} = (w_1, ...w_k)$, to capture the importance among different objectives and is usually set based on the application preference. A further improvement is workload-aware WUN, motivated by our observation that expert knowledge about different objectives is available from the literature. For example, between latency and cost, it is known that it is beneficial to allocate more resources to large queries (e.g. join queries) but less so for small queries (e.g., selection queries). In this case, we use historical data to divide workloads into three categories, (low, medium, high), based on the observed latency under the default configuration. For long running workloads, we give more weight to latency than the cost, hence encouraging more cores to be allocated; for short running workloads, we give more weight to the cost, limiting the cores to be used. We encode such expert knowledge using internal weights, $\boldsymbol{w}^I = (w_1^I, ..., w_k^I)$, and application preference using external weights, $\boldsymbol{w}^E = (w_1^E, ..., w_k^E)$. The final weights are $\boldsymbol{w} = (w_1^I w_1^E, ..., w_k^I w_k^E)$.

VI. Performance Evaluation

In this section, we compare our MOO approach to popular MOO techniques [5], [10], [19], [8] and perform an end-to-end comparison to a STOA performance tuning system, OtterTune [35]. We use DNN models as the default for the MOO experiments as they are among the most time consuming models, and use GP models in the comparison to OtterTune.

Workloads. We used two benchmarks for evaluation.

Batch Workloads: Our batch workloads use the TPCx-BB benchmark [32] with a scale factor 100G. TPCx-BB includes 30 templates, including 14 SQL queries, 11 SQL with UDF, and 5 ML tasks, which we modified to run on Spark. We parameterized the 30 templates to create 258 workloads, with 58 reserved as offline workloads for intensive sampling and 200 as online workloads. We ran them under different configurations, totaling 24560 traces, each with 360 runtime



(a) Uncertain space (job 9, 2d) (b) Frontier of WS and NC (job 9, 2d) (c) Frontier of PF (job 9, 2d)

(d) Uncertain space (job 9, 2d) (e) Frontier of Evo (job 9, 2d) (f) Uncertain space of all 258 jobs

Fig. 4. Comparative results on multi-objective optimization using 258 batch workloads

metrics. These traces were used to train workload-specific models for latency, cost, etc. Feature selection resulted in 12 most important Spark parameters including the number of executors, number of cores per executor, memory per executor, shuffle compress, parallelism, etc. (see [29] for the full list).

Streaming Workloads: We also created a streaming benchmark by extending a prior study [15] on click stream analysis, including 5 SQL templates with UDFs and 1 ML template. We created 63 workloads from the templates via parameterization, and collected traces for training models for latency and throughput. MOO was run on the most important 10 knobs.

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

A. Comparison to MOO Methods

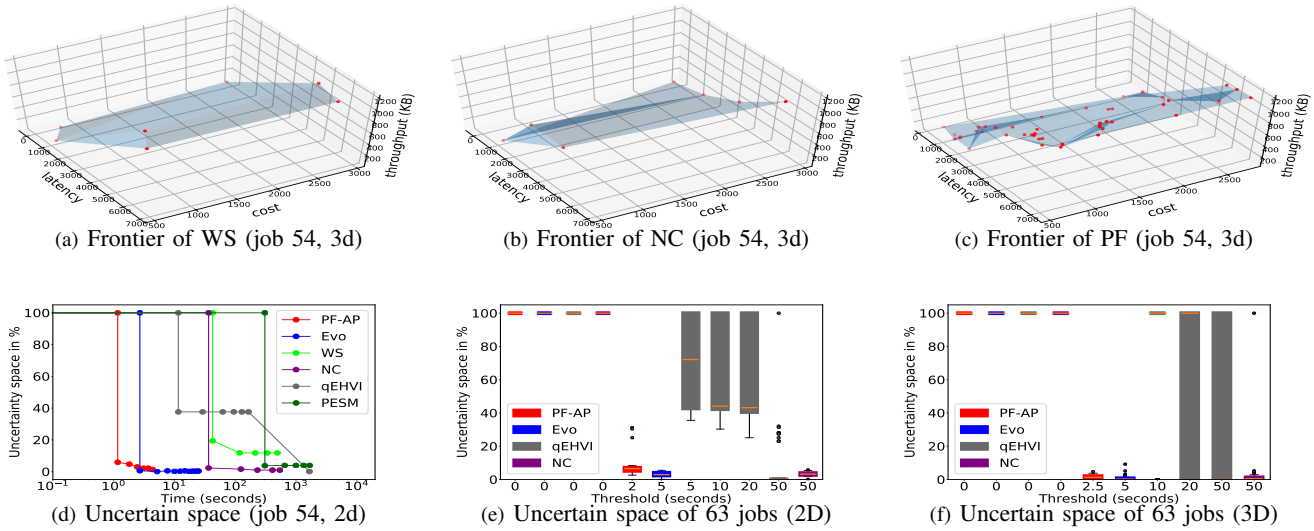
We first compare our PF algorithms, PF-AS and PF-AP, to **five** MOO methods: Weighted Sum (WS) [19], Normalized Constraints (NC) [21], NSGA-II [6] suggested as the most relevant Evolutionary (Evo) method [8], PESM from the Spearmint library [10], and qEHVI from BoTorch [5]. The latter two are Multi-objective Bayesian Optimization (MOBO) methods (see §III). For each algorithm, we request it to generate increasingly more Pareto points (10, 20, 30, 40, 50, 100, 150, 200), which are called probes, as more computing time is invested, except qEHVI that is shown to have best runtime when calling for one point at a time.

Expt 1: Batch 2D. We start with the batch workloads where the objectives are latency and cost (in number of cores). As results across different jobs are consistent, we first show details using job 9. To compare PF-AS and PF-AP to WS and NC, Fig. 4(a) shows the uncertain space (percentage of the total objective space that the algorithm is uncertain about) as more Pareto points are requested. Initially at 100%, it starts to reduce when the first Pareto set (up to 10 points) is produced. We observe that WS and NC take long to run, e.g., ∼47 seconds to produce the first Pareto set. In comparison, PF-AS and PF-AP reduce uncertain space more quickly, with the first Pareto set produced under 1 second by PF-AP. PF-AS does not work as well as PF-AP because as a *sequential* algorithm, its Pareto points found in the early stage have a severe impact on the later procedure, andone low-quality approximate result in the early stage may lead to overall low-quality Pareto frontier. In contrast, PF-AP is a *parallel* algorithm and hence one low-quality approximate result won't have as much impact.

Fig. 4(b) shows the Pareto frontiers of WS and NC created after 47 seconds. WS is shown to have *poor coverage* of the Pareto frontier, e.g., returning only 3 points although 10 were requested. NC generates 8 points, still less than PF-AP shown in Fig. 4(c), which produces 12 points using only 3.2 seconds.

We next compare PF-AP to PESM, qEHVI, and Evo in Fig. 4(d). qEHVI takes 48 seconds to generate the first Pareto set, while PESM takes 362 seconds to do so. This is consistent with common observations that Bayesian optimization can take long to run, hence not suitable for making *online* recommendations by a cloud optimizer. Although Evo runs faster than other prior MOO methods, it still fails to generate the first Pareto set until after 2.6 seconds. Fig. 4(e) shows another issue of Evo: the Pareto frontiers generated over time are *inconsistent*. For example, the frontier generated with 30 probes indicates that if one aims at latency of 6 seconds, the cost is around 36 units. The frontier produced with 40 probes shows the cost to be as low as 20, while the frontier with 50 probes changes the cost to 28. Recommending configurations based on such inconsistent information is highly undesirable.

Fig. 4(f) summarizes the performance of 4 major methods for all 258 workloads, where the x axis is the elapsed time, and the y axis shows the uncertain space across all jobs, with the median depicted by an orange bar. All methods start with 100% uncertain space, and we show when each method starts to reduce until falling below 10% or reaching 100 seconds. PF-AP can generate Pareto sets under 1 second for all jobs, with a median of 8.8% uncertain space, and reduces the median to 5.9% after 2 seconds. Evo remains at 100% within 2 seconds and then achieves a median of 4.2% after 5 seconds. qEHVI can only generate Pareto sets for 54 jobs after 20s, and achieve only 69.4% median after 100s. NC can generate Pareto set for



(a) Frontier of WS (job 54, 3d) (b) Frontier of NC (job 54, 3d) (c) Frontier of PF (job 54, 3d)

(d) Uncertain space (job 54, 2d) (e) Uncertain space of 63 jobs (2D) (f) Uncertain space of 63 jobs (3D)

Fig. 5. Comparative results on multi-objective optimization using 63 streaming workloads

30 jobs after 50s, and achieve 5.8% median after 100s.

Expt 2: Streaming 2D and 3D. We next use the streaming workload under 2 objectives, average latency (of output records) and throughput (the number of records per second), as well as under 3 objectives, further adding cost as the 3rd objective. As results for different jobs are similar, we illustrate them using job 54, while additional results are available in [29]. Fig. 5(a) and Fig. 5(b) show that WS and NC again have poor coverage of the frontier (7 points only each), while Fig. 5(c) shows that PF can better construct the frontier using less time. Evo again returns inconsistent Pareto frontiers as more probes are made, with plots shown in [29].

Regarding running time, Fig. 5(d) confirms that WS, NC, and PESM take long, e.g., 42, 36, and 308 seconds, respectively, to return the first Pareto set, while PF-AP, Evo, and qEHVI are more efficient, taking 1.1s, 2.7s, 11.5s, respectively, to produce the first Pareto set. Fig. 5(e) and 5(f) summarize runtime across 63 workloads for 2D and 3D cases. For 2D jobs, PF-AP is the first to generate Pareto sets, achieving a median of 6.5% under 2 seconds. Evo takes 5 seconds to generate first Pareto sets (which may change at a later time). Both qEHVI and NC need 50 seconds to reduce the median of uncertain space below 10%. The 3D results confirm the same order of the methods in efficiency, with PF-AP taking 2.5 seconds to reduce to 1.3% uncertain space.

B. End-to-End Comparison

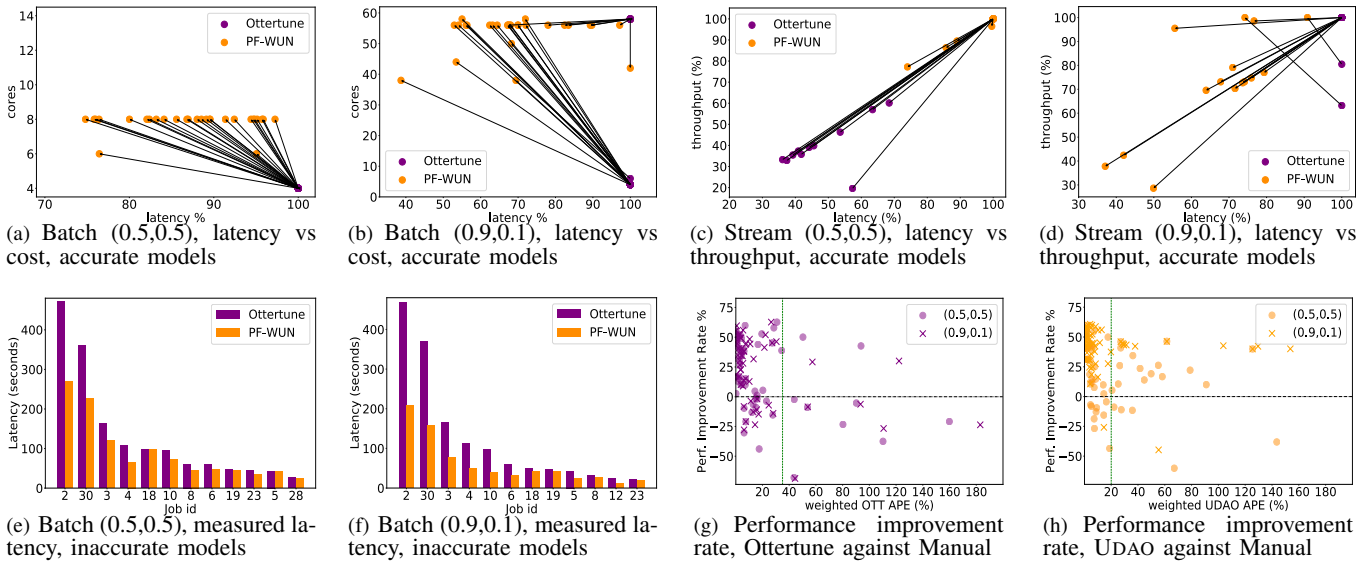
Next we perform an end-to-end comparison of the configurations recommended by our MOO against Ottertune [35]. In this study, since we consider the effect of models, we follow standard ML practice to separate workloads into training and test sets: For TPCx-BB, we created a random sample of 30 workloads, one from each template, as test workloads. We use the traces of other workloads as historical data to train predictive models for the test workloads. Similarly, for the stream benchmark we created a sample of 15 test workloads.

For each test workload, UDAO runs the PF algorithm to compute the Pareto set and then the Weighted Utopia Nearest (WUN) strategy (§V) to recommend a configuration from the set. As Ottertune supports only single-objective (SO) optimization, we apply a weighted method [39] that combines k objectives into a single objective, $\sum_{i=1}^{k} w_i \Psi_i(\boldsymbol{x})$, with $\sum_i w_i = 1$, and then call Ottertune to solve a SO problem. It is known from the theory of Weighted Sum (WS) [19] that even if one tries many different values of $\boldsymbol{w}$, the weighted method cannot find many diverse Pareto points, which is confirmed by the sparsity of the Pareto set in Fig. 5(a) where WS tried different $\boldsymbol{w}$ values.

Expt 3: Accurate models. First, we assume learned models to be accurate and treat model-predicted values of objectives as true values, for any given configuration. For fair comparison, we use the GP models from Ottertune in both systems.

Batch 2D. For 2D batch workloads, Fig. 6(a) shows the performance of the recommended configurations by two systems when $\boldsymbol{w}$=(0.5, 0.5), i.e., the application wants balanced results between latency and cost. Since TPCx-BB workloads have 2 orders of magnitude difference in latency, we normalize the latency of each workload (x-axis) by treating the slower one between PF-WUN and Ottertune as 100% and the faster as a value less than 100%. The number of cores (y-axis) allowed in this test is [4, 58]. For all 30 jobs, Ottertune recommends the smallest number of cores (4), favoring cost to latency. PF-WUN is adaptive to the application requirement of balanced objectives, using 2-4 more cores for each job to enable up to 26% reduction of latency. Fig. 6(b) shows the results for $\boldsymbol{w}$=(0.9, 0.1), indicating strong preference for low latency. For 19 out of 30 jobs, Ottertune still recommends 4 cores as it is the solution returned even using the 0.9 weight for latency. In contrast, PF-WUN is more adaptive, achieving lower latency than Ottertune for all 30 jobs with up to 61% reduction. Further, for 8 jobs, PF-WUN dominates Ottertune in both objectives, saving up to 33% of latency while using fewer cores—in this case, Ottertune's solution is not Pareto optimal.

Streaming 2D. For 15 stream jobs, Fig. 6(c)-6(d) show that for $\boldsymbol{w}$=(0.5, 0.5) the two systems mainly show tradeoffs, while for $\boldsymbol{w}$=(0.9, 0.1), PF-WUN is more adaptive, achieving lower



(a) Batch (0.5,0.5), latency vs cost, accurate models
(b) Batch (0.9,0.1), latency vs cost, accurate models
(c) Stream (0.5,0.5), latency vs throughput, accurate models
(d) Stream (0.9,0.1), latency vs throughput, accurate models
(e) Batch (0.5,0.5), measured latency, inaccurate models
(f) Batch (0.9,0.1), measured latency, inaccurate models
(g) Performance improvement rate, Ottertune against Manual
(h) Performance improvement rate, UDAO against Manual

Fig. 6. Comparative results to Ottertune on single-objective and multi-objective optimization

latency for all jobs with up to 63% reduction of latency.

Expt 4: Inaccurate models. We next consider the case that learned models are not accurate (before enough training data are acquired). For a given objective, our MOGD solver uses the model variance to obtain a more conservative estimate for use in optimization. Further, for TPCx-BB our DNN model [38] offers better latency estimates (20% error rate in Weighted Mean Absolute Percentage Error, where the percentage error is weighted by the objective value) than Ottertune's GP model (35% error rate). In this experiment we use our DNN model to demonstrate the flexibility of our optimizer, while Ottertune can only use its GP model. We consider two cost measures: cost1 in #cores, which is certain; cost2 as a weighted sum of CPU-hour and IO cost, which are both learned models. Thus, cost2 is subject to 15% error using our DNN model and 34% error using Ottertune's GP model.

Next we consider 2D optimization over latency and cost1. For $\boldsymbol{w}$=(0.5, 0.5), we take recommendations from both systems and measure actual latency and cost on our cluster. Fig. 6(e) shows the latency of top 12 long-running jobs. Since both systems use low numbers of cores, the cost plot is omitted here. Notably, to run the full TPCx-BB benchmark, UDAO outperforms Ottertune with 26% savings on running time and 3% less cost. For $\boldsymbol{w}$=(0.9, 0.1), Ottertune's recommendations vary only slightly from (0.5, 0.5), with 6% reduction of total running time, while our recommendations lead to 35% reduction. As Fig. 6(f) shows, UDAO outperforms Ottertune with 49% less total running time, and 48% increase of cost, which matches application's strong preference for latency. For the two long-running jobs (2, 30), Ottertune reports better performance in prediction, but its model is way off and hence in actual running time it achieves much worse results. The prediction and actual latency are more similar in UDAO, enabling better optimization results. Results using latency and cost2 confirm the above observations and are left to [29].

Expt 5: Inaccuracy vs. optimization performance. To quantify the impact of model accuracy on optimization, we collect 120 configurations, recommended by UDAO and Ottertune each, from Expt 4 where optimization was ran using w=(0.5,0.5) or w=(0.9,0.1), cost1 or cost2. We measure actual latency of each configuration and report model accuracy using absolute percentage error (APE) weighted by the latency value. For optimization, we measure *performance improvement rate* (PIR) by comparing the recommended configuration against a manual configuration chosen by an expert engineer.

Fig. 6(g)-6(h) show PIR over weighted APE. 1) The DNN model is more accurate than GP here, with the average error rate shown by a vertical green line. 2) Empirically, we do not observe DNN to be more susceptible to a long tail than GP, since our DNN model is regularized by the L2 loss and also considers variance when running MOGD. 3) When the model accuracy decreases, PIR can degrade to poor values in some cases (around -50%). Overall, Ottertune has more points (38 out of 120) that lead to PIR below 0%, performing worse than the Spark expert, than UDAO (16/120). This is because UDAO adapts better to user preferences for latency and recommends configurations that are more likely to improve PIR.

VII. Related Work

Most relevant work was already discussed in previous sections. Below we survey a few broadly related areas.

Multi-objective optimization for SQL [13], [33], [34] enumerates a finite set of query plans built on relational algebra to select Pareto-optimal ones, which stands in contrast to our need for searching through an an *infinite* parameter space to find Pareto-optimal configurations. MOO approaches for workflow scheduling [13] differ from our MOO in both the parameter space and the solution.

Resource management. TEMPO [31] addresses resource management for DBMSs in the MOO setting: when Service-Level Objectives (SLOs) cannot be all satisfied, it guarantees max-min fairness over SLOs; otherwise, it uses WS for returning a single solution. Morpheus [12] addresses the tradeoff between cluster utilization and job performance predictability by

codifying user expectations as SLOs and enforces them using scheduling methods. WiseDB [17] manages cloud resources based on a decision tree trained on performance and cost features collected from minimum-cost schedules of sample workloads, while such schedules are not available in our case.

Recent performance tuning systems are limited to single-objective optimization and cannot support complex MOO problems. First, platform-specific handcrafted models were developed by leveraging domain knowledge and workload profiling, and then used to solve a single-objective optimization problem [15], [24], [36]. Among search-based methods, BestConfig [40] searches for good configurations by dividing high-dimensional configuration space into subspaces based on samples, but it cold-starts each tuning request. ClassTune [41] solves the optimization problem by classification, which cannot be easily extended to the MOO setting. Among learning-based methods, Ottertune [35] can learn flexible models from data. It builds a predictive model for each query by leveraging similarities to past queries, and runs Gaussian Process exploration to minimize a single objective. CDBTune [39] recommends the best configuration for optimizing the reward (fixed weighted sum of objectives) calculated by Deep RL. To the best of our knowledge, our work is the first to tackle MOO-based performance tuning for big data systems like Spark.

Learning-based query optimization. Recent work [18], [30] uses neural networks to match the structure of optimizer-selected query plans and predicts cardinality, cost, or latency. Neo [16] is a DNN-based query optimizer that bootstraps its optimization model from existing optimizers and then learns from incoming queries. Recent work [28] integrates learned models into a traditional cost-based query optimizer. None of the above work considers MOO like in our work.

VIII. Conclusions

We presented UDAO, a multi-objective optimizer that constructs Pareto-optimal job configurations for multiple task objectives, and recommends a new configuration to best meet them. Using batch and streaming workloads, we showed that UDAO outperforms existing MOO methods [19], [8], [5], [10] in both speed and coverage of the Pareto set, and outperforms Ottertune [35] by a 26%-49% reduction in running time of the TPCx-BB benchmark, while adapting to different application preferences on multiple objectives. In future work, we plan to extend UDAO to support a pipeline of analytic tasks, and incorporate more complex and robust models.

Acknowledgments

This work was partially supported by the European Research Council (ERC) Horizon 2020 research and innovation programme (grant n725561), ARL grant W911NF-17-2-0196, NSF grant 1908536, and China Scholarship Council (CSC). We would like to thank Wei Sheng for the discussion and help in the earlier phase of the project.

References

[1] Amazon EC2 instances. https://aws.amazon.com/ec2/instance-types/.
[2] Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless/.
[3] Bomin: Basic open-source nonlinear mixed integer programming. https://www.coin-or.org/Bonmin/.
[4] Couenne: Convex over and under envelopes for nonlinear estimation. https://projects.coin-or.org/Couenne/.
[5] S. Daulton, et al. Differentiable expected hypervolume improvement for parallel multi-objective Bayesian optimization. arXiv:2006.05078, 2020.
[6] K. Deb, A. Pratap, et al. A fast and elitist multiobjective genetic algorithm: Nsga-ii. Trans. Evol. Comp, 6(2):182–197, Apr. 2002.
[7] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. PVLDB, 2(1):1246–1257, 2009.
[8] M. Emmerich, et al. A tutorial on multiobjective optimization: Fundamentals and evolutionary methods. Natural Computing, 17(3), 2018.
[9] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In ICML, 2016.
[10] D. Hernández-Lobato, et al. Predictive entropy search for multi-objective Bayesian optimization. In ICML, pp. 1492–1501, 2016.
[11] L. iberti. Undecidability and hardness in mixed-integer nonlinear programming. https://www.lix.polytechnique.fr/~liberti/rairo18.pdf. 2018.
[12] S. A. Jyothi, C. Curino, et al. Morpheus: Towards automated SLOs for enterprise clusters. In OSDI, pp. 117–134, 2016.
[13] H. Kllapi, E. Sitaridi, et al. Schedule optimization for data processing flows on the cloud. In SIGMOD, pp. 289–300, 2011.
[14] Knitro user's manual. https://www.artelys.com/docs/knitro/index.html.
[15] B. Li, Y. Diao, and P. J. Shenoy. Supporting scalable analytics with latency constraints. PVLDB, 8(11):1166–1177, 2015.
[16] R. Marcus, P. Negi, et al. Neo: A learned query optimizer. Proc. VLDB Endow., 12(11):1705–1718, 2019.
[17] R. Marcus and O. Papaemmanouil. Wisedb: A learning-based workload management advisor for cloud databases. PVLDB, 9(10):780–791, 2016.
[18] R. Marcus and O. Papaemmanouil. Plan-structured deep neural models for query performance prediction. PVLDB, 12(11):1733-1746, 2019.
[19] R. Marler, et al. Survey of multi-objective optimization methods for engineering. Structural & Multidisciplinary Optimization, 26(6), 2004.
[20] A. Messac. From dubious construction of objective functions to the application of physical programming. AIAA Journal, 38(1), 2012.
[21] A. Messac, et al. The normalized normal constraint method for generating the pareto frontier. Strl. & Multidiscpl. Opt., 25(2), 2003.
[22] Neos guide: Nonlinear programming software. https://neos-guide.org/content/nonlinear-programming.
[23] Oracle Cloud Data Flow. https://www.oracle.com/big-data/data-flow/.
[24] K. Rajan, D. Kakadia, et al. Perforator: eloquent performance models for resource optimization. In SoCC, pp. 415–427, 2016.
[25] S. Ruder. An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747, 2016.
[26] J. Snoek, et al. Practical Bayesian Optimization of Machine Learning Algorithms. NIPS, 2012.
[27] E. Schulz, et al. A tutorial on gaussian process regression: Modeling, exploring, and exploiting functions. J. of Math. Psych., 85:1-16, 2018.
[28] T. Siddiqui, A. Jindal, et al. Cost models for big data query processing: Learning, retrofitting, and our findings. In SIGMOD, pp. 99–113, 2020.
[29] F. Song, K. Zaouk, et al. Boosting cloud data analytics using multi-objective optimization. http://scalla.cs.umass.edu/papers/udao2020.pdf.
[30] J. Sun and G. Li. An end-to-end learning-based cost estimator. Proc. VLDB Endow., 13(3):307–319, 2019.
[31] Z. Tan and S. Babu. Tempo: robust and self-tuning resource management in multi-tenant parallel databases. PVLDB, 9(10):720–731, 2016.
[32] TPCx-BB benchmark for big data analytics. http://www.tpc.org/tpcx-bb/
[33] I. Trummer and C. Koch. Approximation schemes for many-objective query optimization. In SIGMOD, pp. 1299–1310, 2014.
[34] I. Trummer and C. Koch. An incremental anytime algorithm for multi-objective query optimization. In SIGMOD, pp. 1941–1953, 2015.
[35] D. Van Aken, A. Pavlo, et al. Automatic database management system tuning through large-scale machine learning. In SIGMOD, 2017.
[36] S. Venkataraman, Z. Yang, et al. Ernest: Efficient performance prediction for large-scale advanced analytics. In NSDI, 2016.
[37] M. Zaharia, M. Chowdhury, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In NSDI, 2012.
[38] K. Zaouk, F. Song, et al. UDAO: A next-generation unified data analytics optimizer (vldb 2019 demo). PVLDB, 12(12):1934–1937, 2019.
[39] J. Zhang, Y. Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In SIGMOD, 415–432, 2019.
[40] Y. Zhu, J. Liu, et al. BestConfig: tapping the performance potential of systems via automatic configuration tuning. In SoCC, 338–350, 2017.
[41] Y. Zhu, J. Liu, et al. ClassyTune: A Performance Auto-Tuner for Systems in the Cloud. IEEE Trans, on Cloud Computing, 1-1, 2019.