UDAO: A Next-Generation Unified Data Analytics Optimizer

Khaled Zaouk, Fei Song, Chenghao Lyu, Arnab Sinha, Yanlei Diao, Prashant Shenoy

UDAO: A Next-Generation Unified Data Analytics Optimizer

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

ABSTRACT

Big data analytics systems today still lack the ability to take user performance goals and budgetary constraints, collectively referred to as "objectives", and automatically configure an analytic job to achieve the objectives. This paper presents UDAO, a unified data analytics optimizer that can automatically determine the parameters of the runtime system, collectively called a job configuration, for general dataflow programs based on user objectives. UDAO embodies key techniques including *in-situ modeling*, which learns a model for each user objective in the same computing environment as the job is run, and *multi-objective optimization*, which computes a Pareto optimal set of job configurations to reveal tradeoffs between different objectives. Using benchmarks developed based on industry needs, our demonstration will allow the user to explore (1) *learned models* to gain insights into how various parameters affect user objectives; (2) *Pareto frontiers* to understand interesting tradeoffs between different objectives and how a configuration recommended by the optimizer explores these tradeoffs; (3) *end-to-end benefits* that UDAO can provide over default configurations or those manually tuned by engineers.

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

1. INTRODUCTION

Today's big data analytics systems remain best effort in nature: despite wide adoption, most of them lack the ability to take user performance goals or budgetary constraints, collectively referred to as "objectives", into account in order to automatically configure an analytic job to achieve those objectives. Consider a user that aims to run a mix of analytics tasks on EC2 instances of AWS. Common practice today involves the following manual effort from the user:

First, the user needs to choose from over 60 Amazon EC2 instance types that differ in the number of cores and memory available. For example, the number of cores determines the degree of parallelism that can be achieved in analytics and strongly affect the performance and cloud costs. The appropriate choice depends on the user workloads and objectives, and is often an educated guess by the user. Once the cloud instance type is chosen, the user may still need to tune the parameters of the runtime system, collectively referred to as a "job configuration", for each analytical task in order to achieve good performance. In case of the popular Spark platform, these runtime parameters include *parallelism* (for reduce-style transformations), *Rdd compression* (boolean), *Memory per executor*, *Memory fraction* (of heap space), *Batch interval* (the size of each minibatch) and *Block interval* (the size of data handled by a map task) in the streaming setting, to name a few.

Choosing the right cluster and job configuration to meet user objectives is a difficult task. A recent study [6] shows that even for HiveQL queries, expert engineers were often unable to make the correct choice between two cluster options, and their estimated runtime ranged from 20x underestimation to 5x over-estimation. Searching for the configuration that best suits the user objectives is largely a trial-and-error process today, including manual tuning of a large number of runtime parameters as well as changes to cloud instance types with more cores or memory. The above discussion indicates that cloud computing for data analytics today largely guarantees availability, but lacks the ability to take user objectives as service level agreements (SLAs).

In this paper, we present the design and demonstration of a next-generation **Unified Data Analytics Optimizer** (UDAO) that takes as input a user analytical task in the form of a dataflow program (which subsumes SQL queries) and a set of user objectives, and produces as output a cluster configuration with a suitable number of cores as well as other parameters of the runtime system that best meet the user objectives. Our UDAO system has several distinct features:

1. *Unified analytics*: Our optimizer supports broad analytics including SQL queries, machine learning tasks, etc., that represent the general paradigm of dataflow programs. This design decision is informed by our discussions with two cloud service providers, detailed in the next section, showing that analytics today makes intensive use of user-defined functions for ETL and machine learning tasks. Our approach to optimization does not distinguish between different types of analytics but instead relies solely on obser-

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352103

vations of runtime behaviors of such analytical tasks and represents a "blackbox" approach to unified analytics.

2. *In-situ modeling of each user objective.* To enable optimization, the UDAO system builds a performance model for each objective of a user task. Performance modeling in this new environment involves major challenges such as diverse user objectives (e.g., throughput, latency, resource utilization), heterogenous hardware, and complex, dynamic system behaviors. Our system takes a new approach that learns a model for each user objective in the same computing environment where the user task is being executed, which we refer to as *in-situ modeling*.

3. *Multi-objective optimization.* A key feature of UDAO is multi-objective optimization (MOO), which takes a set of user objectives and constructs a Pareto (optimal) set of job configurations, where a job configuration belongs to the Pareto set if it is not dominated by any other job configuration in all objectives. The visualization of this set, called a Pareto frontier, illustrates the tradeoffs between different objectives, and allows the system to automatically recommend a job configuration that explores the tradeoffs in a manner that best suits user needs.

In this demonstration, we will illustrate UDAO as follows: (1) **Benchmarks**: we have designed multiple benchmarks based on the needs specified by two cloud service providers and collected large traces to enable the demonstration. (2) **Learned models**: by visualizing the model learned for each user objective, UDAO allows the user to gain insights into how various parameters, including their complex interactions, affect user objectives. (3) **Pareto frontiers:** by visualizing a Pareto frontier, UDAO allows the user to understand interesting tradeoffs between objectives and how a configuration recommended by the optimizer explores these tradeoffs. (4) **End-to-end benefits**: combining modeling and MOO, we demonstrate the net benefits UDAO offers compared to default configurations and those manually tuned by engineers. (5) **Comparative results**: our demo also includes comparisons to alternative modeling methods such as Ottertune [8] and popular MOO methods such as [1, 4, 5] to illustrate their strengths and limitations.

2. SYSTEM OVERVIEW

In this section, we present the constraints and needs from real-world use cases and our corresponding system design.

2.1 Requirements of Real-World Use Cases

We model an analytic task as a *dataflow* program as commonly used in systems such as Spark and Flink. For execution, the system transforms the program to a *cluster execution plan* with runtime parameters instantiated. As stated before, these parameters control the degree of parallelism, granularity of scheduling, memory allocated to executors and data buffers, compression options, shuffling strategies, etc. When the plan is executed, we call it a *job* and refer to the runtime parameters collectively as the *job configuration.*

We held extensive discussions with two cloud service providers (anonymized for confidentiality reasons) and summarize their constraints and real-world needs as follows.

1. *Mixed workloads.* Besides SQL queries, analytics pipelines today often run large ETL jobs for data cleaning, transformation, and integration, as well as machine learning tasks for deep analysis. Given such a mix, a *blackbox* approach that is able to handle a variety of workloads without making specific assumptions (e.g., availability of query plans) is a good starting point.

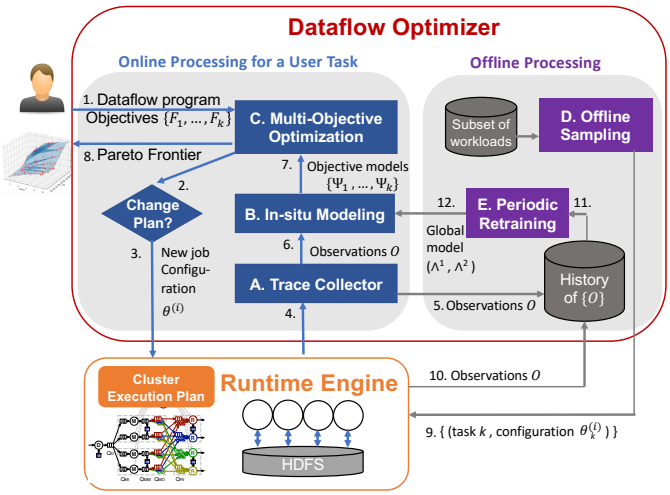


Figure 1: Overview of a dataflow optimizer.

2. *Repeated workloads.* An optimizer is helpful when analytical tasks are long running and repeated frequently, e.g., daily and hourly batch jobs. 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 rolled into the batch job, the streaming job is periodically restarted over new data.

3. *Similarities across workloads.* Similarities across workloads arise for several reasons: (a) Early processing tasks in analytics pipelines are often similar; they can be shared by tens of downstream tasks. (b) Most workloads are parameterized, i.e., generated from a set of templates with the parameters set to appropriate values by each user. Such similarities offer an opportunity for the optimizer to improve prediction accuracy, even under a blackbox approach.

4. *Private cloud.* Our discussions with cloud service providers have led us to focus on private clouds, where the service provider offers support to a major customer or its internal analytics groups—optimization in such settings is more tractable than in the public cloud and is also more important for large customers. In this setting, we assume that it is possible to gain access to a subset of user workloads (e.g., 10%-20%) in order to tune overall performance. The implications on the design of the optimizer include: (*a*) Exploration over the configuration space is possible via **offline** sampling by the optimizer over a subset of workloads. (*b*) When production workloads are running online, sampling over different configurations is not possible; instead, the goal of the optimizer is to recommend new configurations to achieve desired performance as quickly as possible.

2.2 System Design

The above requirements lead to our system design as shown in Figure 1. The top part of the figure depicts the dataflow optimizer, while the bottom part depicts the runtime engine.

Inside the optimizer, the left panel shows the **online** path when a user job is submitted. The user provides her dataflow, objectives $(F_1, \ldots, F_k)$, and optionally value constraints on these objectives, $[F_i^L, F_i^U]$ (see the edge labeled as step 1). The job initially runs using a default or user-specified con-

figuration, θ^1 (i.e., a trivial choice for steps 2 and 3). For the new job, the optimizer proceeds in three major phases.

A. Trace collection: This phase collects traces, collectively called *observations*, during job execution (step 4), which include: (*i*) user objectives such as throughput, latency, resource utilization, and computing cost; (*ii*) application-level metrics, e.g., from Spark in our prototype, collecting the records read and written, bytes read and written, bytes spilled to disk, fetch wait time, etc.; (*iii*) OS-level metrics such as CPU, IO, and network usage. This trace data is written to a database as being collected (step 5).

B. In-situ modeling: If the job is seen the first time, trace collection will be run for a period of time (e.g., 10 minutes). Then in-situ modeling is activated (step 6). It takes as input (*i*) the traces for the new job, and (*ii*) a global deep learning model, $\Lambda = (\Lambda^1, \Lambda^2)$, trained offline using all past jobs (to be detailed shortly). In-situ modeling performs two tasks: (1) It transforms the new traces into a numerical vector and runs it through the *encoding model* Λ^1 to derive a formal description (*encoding*) of the current job, denoted by W. (2) It then feeds W to global *regression models* Λ^2 to build job-specific predictive models, $(\Psi_1, \ldots, \Psi_k)$, one for each user objective. It is important to note that **no** training is incurred for the new job on the online path.

C. Optimization: Given the job-specific predictive models, the multi-objective optimization (MOO) module searches through the space of configurations and computes a set of Pareto-optimal configurations for the job (step 7). The system then returns a visualized surface of these configurations, called the Pareto frontier, to the user (step 9). A configuration that best explores the tradeoffs among different objectives is then chosen (step 2 repeated), and this new configuration, θ^2, is recommended for future execution of the job (step 3 repeated).

When the user job runs the next time, the system repeats steps 4, 5, 6, and 7. If the global model was retrained between the last and current execution, in-situ modeling runs again; otherwise, the previous predictive models are still valid. For optimization, if the job-specific predictive models are updated, the Pareto-frontier will be recomputed; otherwise, the previous one will be reused. If the user wants to adjust her preference on the objectives (e.g., from favoring low latency to high throughput), she can indicate so by setting the bounds, $[F_i^L, F_i^U]$, on her objectives differently. Given the Pareto frontier, the MOO module can quickly return a new configuration, θ^3, that suits the new objectives.

The right panel of the optimizer shows the **offline** processing with two major tasks:

D. Offline sampling: The optimizer uses a small subset of user workloads for offline sampling over the configuration space, e.g., exploring different configurations and observing the related objectives to enable more accurate models (steps 9 and 10). Our work uses a mix of Bayesian Optimization, a sequential optimization technique for exploring an unknown space, and heuristics based sampling, leveraging domain knowledge of configurations (e.g., from Spark best practices) to overcome the cold start problem.

E. Periodic retraining: The optimizer periodically retrains by taking observations from all past online jobs and jobs sampled offline, returning an updated global model $\Lambda^t = (\Lambda^{t1}, \Lambda^{t2})$ (steps 11 and 12). The frequency of retraining can be set to daily or when the predictive models of some online jobs are observed to be not accurate enough.

2.3 Key Techniques

We next discuss the key techniques employed in UDAO.

Modeling. For each dataflow program, we model each user objective as a function over all tunable parameters of the runtime system. Learning such a model for each user objective and a specific cluster environment has the potential to adapt to different objectives, hardware, and software characteristics, while static models [2, 3, 6] often fail to adapt due to hard-coded function shapes and constants.

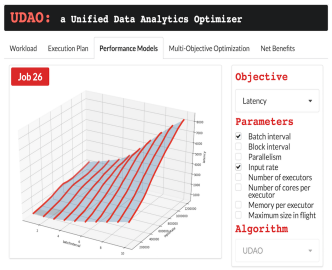
However, since dataflow programs can use arbitrary methods and be written in any program language, it is difficult for the optimizer to understand the nature of the computation, raising a major challenge in building accurate models. Ottertune [8], a state-of-the-art modeling technique for black-box programs, employs *workload mapping*: it tries to map a new job to the "nearest" past job based on the similarity of runtime observations for common configurations, and then uses the information about that past job to predict the performance of the new job. This approach, however, is not always effective in our problem setting because the majority of user jobs are run with a small number of configurations (e.g., 4-5) out of numerous possible configurations. Hence, a new job and its nearest past job may not have many (or any) configurations in common, rendering the mapping not effective. Our work explores the combination of two ideas.

1) *Representation learning*: Deep learning (DL) is known for its power of representation learning, which allows us to perform layers of non-linear transformation of runtime observations to extract a representation W that (*i*) characterizes the fundamental aspects of the work being done, called a *workload encoding*; (*ii*) remains invariant when different configurations are used for the workload. Given W, we can then build a regression model that predicts a user objective based on W and a specific configuration used. In our work, we devise customized neural networks to learn both the encoding and regression models, denoted as $\Lambda = (\Lambda^1, \Lambda^2)$. Unlike Ottertune which trains a separate model for each job, our models are trained using *all* past jobs, and can exploit commonalities that exist across multiple workloads. Such models can be quickly customized for each new job to extract its own encoding and predictive models.

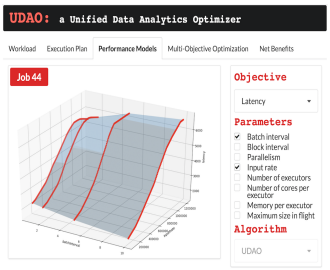
2) *Workload mapping*: As an optimization, when a new job arrives, we do not trigger expensive training using the traces of the new job. Instead, we perform workload mapping from the new job to a past one using their encodings, and leverage the trained regression model for the old workload to predict performance for the new one. Since the encodings are invariant to the configurations used, our mapping does not require the new and old workloads to have many configurations in common. Later as the traces from the new job are included in periodic retraining, the encoding model becomes more accurate and improves future workload mapping.

Multiple-objective optimization. Our multi-objective optimization (MOO) problem is different from that for SQL queries [7] in that MOO for SQL examines a finite set of query plans using a *combinatorial* approach, while our system needs to search through a potentially infinite set of values of runtime parameters, which are often numerical, and hence uses a *numeric* approach.

In addition, we pose two requirements on the computation of the Pareto-frontier to ensure effectiveness and efficiency: 1) *Progressive expansion* of the Pareto frontier (akin to the loading of a Google earth image) so that the optimizer offers

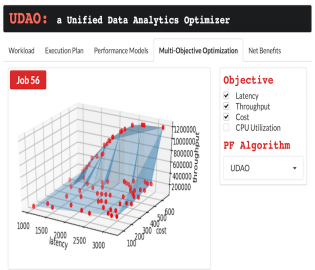


(a) BatchInterval vs. InputRate (job 26)

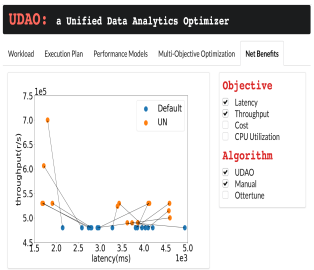


(b) BatchInterval vs. InputRate (job 44)

Figure 2: Learned models for latency



(a) PF of UDAO: latency, throughput, cost



(b) End-to-end benefits of UDAO

Figure 3: Pareto frontiers (PF) and net benefits

correct information soon after a job is submitted, and refines the frontier with more details as more computation time is invested; 2) *Incremental computation* so that a Pareto frontier with more details reuses the computation incurred previously. Popular MOO techniques in the literature fail to meet the new requirements in our problem setting: in particular, evolutionary algorithms [1] violate requirement 1, while Weighted Sum [4] and Normal Constraints [5] violate requirement 2. Our system uses a customized method to transform a MOO problem to a set of constrained optimization problems to find an increasingly larger set of Pareto-optimal solutions, meeting both requirements above.

3. DEMONSTRATION

Our demo showcases UDAO developed on top of Apache Spark, such that the user can understand *i*) the difficulty of manual parameter tuning, and the effect of these parameters on user objectives as indicated by the automatically learned models; *ii*) the tradeoffs between user objectives, and how a system recommended configuration explores these tradeoffs; *iii*) end-to-end benefits that UDAO can provide over manual performance tuning; *iv*) comparative analysis between our techniques and alternative techniques. Specifically, our demonstration has the following modules:

1. Benchmarks. We developed two benchmarks based on the needs specified by two cloud service providers. (a) Big-Bench (TPCx-BB) for batch analytics includes 30 workloads, which can be divided into 14 SQL tasks, 11 SQL with UDFs and 5 ML workloads. (b) We also designed a new stream benchmark by extending previous workloads on click stream analysis [3] to include stream SQL queries, SQL+UDF queries, and machine learning tasks. As suggested by our industry collaborators, these workloads have been parameterized in different ways to control the similarities among workloads. We collected traces from running these benchmarks for weeks to enable the demonstration.

2. Learned models. In this module, our system provides the original program, the execution plan, and all statistics available on the Spark UI for the user to consider. By visualizing the model learned for the same objective, UDAO allows the user to gain insights into the important parameters, including complex interactions between them, and understand how they affect latency. For example, Fig. 2(a) shows the effect of BatchInterval in relation to InputRate, where a smaller value of BatchInterval is always preferred, while Fig. 2(b) shows that BatchInterval can exhibit opposite trends: a larger value reduces latency when the input rate is low, but increases latency when the input rate is high.

Besides visualizing the model, we also show the configuration chosen by UDAO and minimum latency achieved.

3. Pareto frontiers. The next module addresses multiple objectives such as latency, throughput, and computing cost. By visualizing a Pareto frontier, UDAO allows the user to visually understand tradeoffs between the objectives. As Fig. 3(a) shows, in some regions by comprising a little on latency, the system can achieve much higher throughput, hence worth considering by the user. We also show how a configuration recommended by the optimizer, using a set of strategies developed internally, explores these tradeoffs.

4. End-to-end benefits. By combining modeling and MOO, we show the net benefits that UDAO provides for user objectives compared to default configurations or those manually tuned by engineers. For example, Fig. 3(b) shows latency and throughput of 15 jobs under the configurations manually chosen by our engineer (the blue dots) and those chosen by our optimizer (orange dots). The performance of the new configuration dominates the manual configuration in 9 jobs, while it explores tradeoffs in the rest 5 jobs.

5. Comparative results. In modules 2-4, our demo also includes a comparison to alternative modeling methods such as Ottertune [8] and popular MOO methods such as Weighted Sum [4], Normal Constraints [5], and evolutionary algorithms [1], to illustrate their strengths and limitations.

Acknowledgements: This work was funded by European Research Council (ERC) Horizon 2020 research and innovation programme (grant n725561), the US National Science Foundation grant 1836752, and a research gift from the Université Paris Saclay.

4. REFERENCES

[1] M. T. Emmerich and A. H. Deutz. Multi-objective optimization: Fundamentals and evolutionary methods. *Natural Computing*, 17(3):585–609, Sept. 2018.

[2] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[3] B. Li, Y. Diao, and P. J. Shenoy. Supporting scalable analytics with latency constraints. *PVLDB*, 8(11):1166–1177, 2015.

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

[5] A. Messac, A. Ismailyahaya, and C. A. Mattson. The normalized normal constraint method for generating the pareto frontier. *Structural and Multidisciplinary Optimization*, 25(2):86–98, 2003.

[6] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan. Perforator: eloquent performance models for resource optimization. In *SoCC*, 415–427, 2016.

[7] I. Trummer and C. Koch. An incremental anytime algorithm for multi-objective query optimization. In *SIGMOD*, 1941–1953, 2015.

[8] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, 1009–1024, 2017.