Towards Optimal Resource Allocation for Big Data Analytics

Anish Pimpley
Microsoft
anish.pimpley@microsoft.com

Shuo Li
Microsoft
shuol@microsoft.com

Rathijit Sen
Microsoft
rathijit.sen@microsoft.com

Soundararajan Srinivasan
Microsoft
soundararajan.srinivasan@microsoft.com

Alekh Jindal*
Keebo Inc
alekh@keebo.ai

ABSTRACT

Optimizing resource allocation for analytical workloads is vital for reducing operational costs in modern cloud-oriented query processing services. At the same time, it is incredibly hard for users to allocate resources per query in the newer breed of Big Data processing systems, and they frequently end up mis-allocating by orders of magnitude. While prior work has focused on predicting peak resource allocation, it misses opportunities for more aggressive resource allocation to trade-off resource savings with query performance. Additionally, these methods fail to predict resource allocations for queries that have not been observed in the past.

In this paper, we tackle both these problems. We present a system for optimal resource allocation in big data systems that can predict performance impact for candidate resource allocations, for both new and past observed queries. We introduce the notion of a performance characteristic curve (PCC) as a parameterized representation that can compactly capture the relationship between resources and performance. To tackle the challenge of training data sparsity, we introduce a novel data augmentation technique to efficiently synthesize the entire PCC using a single run of the query. Lastly, we demonstrate the advantages of a constrained loss function coupled with GNNs, over traditional ML methods, for capturing the domain specific behavior through an extensive experimental evaluation over SCOPE big data workloads at Microsoft. Our results show that our fast and novel skyline simulation technique for data augmentation is sufficiently accurate, with a median percentage error in run time estimates of 9% on past jobs and our ML models can estimate the run time of jobs for different token counts well, before they have executed, with a median percentage error of 39% or less.

1 INTRODUCTION

Modern cloud computing has alleviated the need for dedicated resources [25]. For query processing, this means that Big Data query processing systems, such as SCOPE[12], Athena[1], and Big Query[38], can dynamically allocate resources for each incoming query without requiring users to reserve a dedicated capacity for their applications. However, allocating resources efficiently is a challenge in these systems. This is because big data queries have a complex relationship between resource allocation and performance, that is very hard to reason about, and that relationship changes over the course of query execution [26].

*Work done while at Microsoft

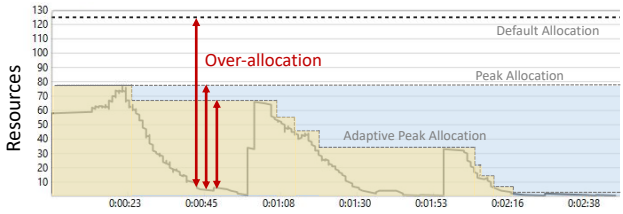


Figure 1: Skyline of resources (tokens) used by a SCOPE job and over-allocations with different allocation policies.

Figure 1 illustrates the challenge in per-query resource allocation. It shows the amount of resources used by a SCOPE query (called *job*), over the duration of its execution on the Cosmos Big Data analytics platform at Microsoft [12], and the resources allocated to the job under various allocation policies. Reusing terminology from prior work [26, 32], we call this time series representation of the job's resource usage as the *skyline* of the job. Similar to prior work [9, 34], we use the term *token* to denote a unit of resource allocation for SCOPE jobs on Cosmos, with *token* being synonymous to *container* and *resource* in this work. Although this example job uses less than 80 tokens, the job was allocated 125 tokens by default, as shown by the dashed line marked as 'Default Allocation'. Thus, the job is over-allocated, with the extent of over-allocation varying as the job runs depending on the number of tokens actually used by the job at that point in time. As can be seen from this example skyline, it is incredibly hard for a user to anticipate such a complex resource usage behavior and allocate resources accordingly.

Recent works have restricted to either predicting the peak resource use [34], thereby aiming to achieve a 'Peak Allocation' policy as shown in Figure 1, or to adaptively giving up non-required resources over the course of query execution [9], thereby aiming to achieve an 'Adaptive Peak Allocation' policy as shown in the figure. Although these policies reduce the over-allocation compared to that with 'Default Allocation' for this job, this is not enough because we see numerous valleys in the job skyline which consequently result in over-allocations at those points. Therefore, allocating for the peak resource use, either upfront or adaptively over time, is highly *conservative* and misses valuable opportunities for operational efficiency. With a more *aggressive* allocation policy, one could allocate less-than-peak resources *provided that the resulting performance loss (if any) is within an acceptable limit.* But implementing such a policy requires the ability to predict a job's expected performance impact, upfront, for a set of candidate resource allocations so that the *optimal* allocation, i.e., the minimum resources that satisfy the user-specified performance constraints, can be chosen for the job.

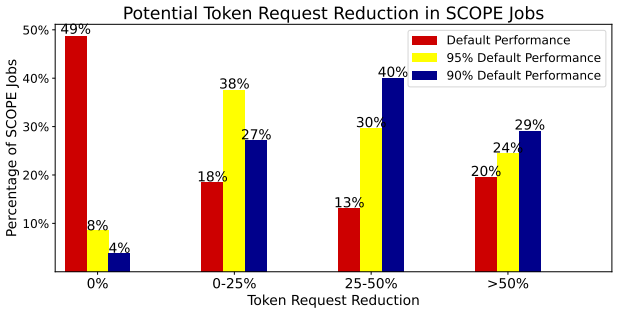


Figure 2: Bar plot of potential token request reduction in production SCOPE jobs at Microsoft.

Figure 2 shows the potential reduction in the number of tokens requested by SCOPE jobs in different scenarios. We see that 51% of the SCOPE jobs could request fewer tokens without any estimated impact on performance (the accumulated token request reduction greater than 0%, i.e., the second, third and fourth red bars), with 33% of the jobs requiring less than 75% of the tokens (the accumulated token request reduction greater than 25%, i.e., the third and fourth red bars), and 20% of the jobs requiring less than 50% of the tokens (token request reduction greater than 50%, i.e., the fourth red bar). Additionally, if the users are willing to accept a 5%-10% performance loss, then the percentage of jobs that could utilize fewer tokens go up to 92%-96%, with 24%-29% of the jobs requiring less than 50% tokens (the yellow/green bars).

Utilizing fewer tokens reduces job wait time and improves the overall resource availability for other jobs in the cluster [34]. Interestingly, our conversations with internal Microsoft users reveal that they do have the above intuition and are in fact looking to allocate resources below the overall peak. However, it is very hard for them to manually estimate the optimal resources required for different jobs [32]. To understand further, we ran a qualitative user study with 12 experienced engineers who have been using SCOPE for a while. Interestingly, only 5 of them demonstrated an understanding of what token allocation means, and yet all 5 either guessed or selected the default value when deciding the number of tokens to allocate per SCOPE job.

In this paper, we present a machine-learning (ML) based approach to predict, at compile-time, the optimal resource allocation for each query in a big data query processing system. Our key observation is that the relationship between resource allocation and query performance could be approximated using an exponentially decaying curve, as shown in Figure 3 and referred to as *performance characteristic curve* (PCC) henceforth in this paper. This is because while job performance improves significantly with more resources (tokens) initially, the change is diminishing for larger token counts. Our goal then is to learn the PCC for all jobs, i.e., given the compile time characteristics or features of a job, predict the parameters for its PCC. Such a curve can be used for both point prediction, i.e., predict the job run time given a token amount, and trend prediction, i.e., predict the job run times for each point in a range of token counts. The monotonicity property of PCC is further useful in helping users understand the proposed model and the performance/resource trade-offs, allowing them to tune the resource allocation based on their acceptable performance range and service-level objectives (SLOs).

Challenges. The biggest challenge in learning the PCC for all jobs in the workload is the limited historical data available for training. Queries in the historical workloads were executed with a given resource allocation, i.e., a single token count, while we need to gather performance changes across different token counts in order to learn the PCC model. The problem becomes worse with the high diversity and the massive scale in typical enterprise workloads that are hard to model, e.g., SCOPE processes hundreds of thousands of jobs per-day over petabytes of data from several different business units across the whole of Microsoft, making it non-trivial to train PCC at such complexity and scale. One option could be to use a job's most recent resource allocation skyline to estimate the PCC, however, the skyline could change significantly over time due to changes in workloads, such as changes in the input sizes. Furthermore, newer and ad-hoc jobs with no historical data do not have historical skylines. Finally, unlike prior approaches for learning cost models [37], the PCC model needs to capture the diminishing nature of performance improvements as more resources are allocated, i.e., not just point predictions but also trends predictions to understand and reason about the trade-offs.

Contributions. We address the above challenges in this paper[1] and make the following key contributions.

- We present TASQ (**T**oken **A**llocation for **S**calable **Q**ueries), an end-to-end ML pipeline to predict optimal token counts in SCOPE like query processing systems. (Section 2)
- We introduce an efficient data augmentation technique for enriching sparse training data using a job skyline simulator, coined AREPAS, that can accurately synthesize skylines with alternate token counts. (Section 3)
- We describe an ML approach for point and trend predictions using a rich set of features from past job graphs. Our model characterizes the Performance characteristic curve (PCC) using two parameters of a power-law curve and predicts these parameters for a given arbitrary job. (Section 4)
- We present extensive evaluation results of TASQ incorporating different ML approaches. Our results show that an XGBoost-based approach has the best accuracy for point prediction. However, its trend prediction is not always correct. In contrast, feed forward neural networks and graph neural networks guarantee a monotonically non-increasing trend between resources allocation and job run time, and both models have reasonably good accuracy for point prediction with the median absolute error in run time prediction being 39% or less over a large production SCOPE workload at Microsoft. (Section 5)

2 TASQ OVERVIEW

We study the problem of efficient resource allocation in the context of SCOPE [12], which powers the internal big data workloads throughout the whole of Microsoft. SCOPE runs hundreds of thousands of jobs that process petabytes of data per-day, with each SCOPE job being a DAG of operators that gets executed using up to thousands of containers (referred to as tokens) in parallel. SCOPE users submit their jobs along with a desired token amount that is allocated as guaranteed resources before the job can start. Unfortunately, users rarely make informed decisions on the appropriate number of tokens needed for their jobs, and they typically opt for the default values, i.e., a static number that is clearly suboptimal for different jobs. Poor token allocation leads to performance loss, higher wait times, and more operational costs for customers. Our recent work considers peak resources, either upfront or adaptively over time [9, 34]. However, peak

[1] An early version of this work is available as a preprint [31].

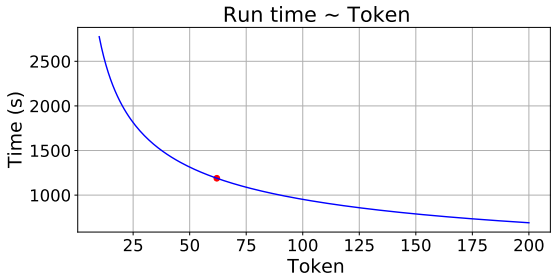


Figure 3: Trade-off between allocated resources and job run time. Red marker indicates an elbow in the curve.

allocation is utilized only for a small portion of a job's lifetime. Instead, we could allocate tokens below the peak usage without incurring a noticeable loss in performance. We refer to this maximal allocation under the peak, that optimally trades off resource cost and run time, as the optimal resource allocation.

2.1 Problem Definition

We now define the optimal resource problem. Let J be an analytical job and A be the resources (token count) allocated to it. Also, let R be the performance, measured typically in terms of the total run time of job J. Our goal then is to learn the performance characteristic curve of job J, i.e., its run time as a function of resource allocation, as illustrated in Figure 3. Formally,

$$PCC_J : R = f(A) \tag{1}$$

Given a PCC_J, we can then find the optimal resource allocation using gradient descent with a termination condition, i.e., a threshold beyond which the gains in performance are diminishing. This threshold could be controlled by the users or administrators, e.g., asking a minimum 1% performance improvement for every additional token count that is allocated.

2.2 System Implementation

TASQ is implemented within the broader workload optimization platform that we have been building at Microsoft [24]. Figure 4 shows the end-to-end TASQ pipeline for training and scoring of PCC prediction models. The TASQ training pipeline ingests historical data from the job repository that includes query plans, runtime characteristics, and other job metadata, and transforms them into a clean model-suitable tabular format, which it stores on Azure Data Lake Storage (ADLS) [2]. Thereafter, TASQ runs the featurization step and trains the models on Azure Machine Learning (AML) [4]. It then registers the trained model in AML model store and deploys it as an Azure Kubernetes Service (AKS) [3]. This service endpoint is used to integrate with a Python client for SCOPE. For an incoming SCOPE job, that is submitted through a client submission system, the TASQ scoring pipeline obtains compile-time job information from the SCOPE compiler, featurizes it, and passes it to the deployed model service for predicting the PCC for that job. Based on the configuration, the system may either directly pass the predicted optimal token count to the job scheduler along with the job for execution or display the PCC to the users for them to understand the performance-resource trade-off and to make an informed decision about the token count.

As discussed before in Section 1, the biggest challenge in learning the PCC is the limited training data on different token counts.

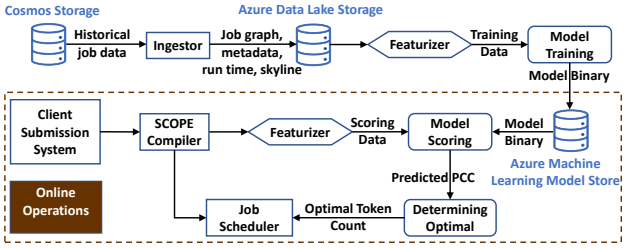


Figure 4: TASQ system integration.

Therefore, TASQ uses a novel simulator, called AREPAS, to generate augmented training input. Given a job's resource consumption skyline, AREPAS can synthesize the run time for the same job with different token allocations. We utilize AREPAS to generate synthetic data that augments the historical dataset. To train the model, we characterize the output of the simulator by distilling it down to two parameters that define a power-law shaped PCC curve. Our training setup uses these parameters as targets for the ML model. This implicitly constraints the model and guides it to learn relationships that reflect known intuitions of domain experts. To accurately learn these parameters, we construct a loss function containing two components which are balanced by tuned weights, namely mean absolute error of PCC parameters and mean absolute error of run time prediction.

In the rest of the paper, we first describe training data augmentation in Section 3, then we discuss the prediction models in Section 4, and show the evaluation results in Section 5.

2.3 Applicability to Other Platforms

The ideas in TASQ can be applied to decide resource allocations for other platforms as well. In fact, in our companion work, we have followed up on TASQ to automatically determine the optimal number of executors for Spark SQL queries [35, 36]. We briefly differentiate the general aspects of TASQ from its platform-specific adaptations below.

The general aspects are the concept of the PCC, modeling it with mathematical functions, using ML models to learn and predict the relationship between query plan characteristics and query inputs to the PCC parameters, using compile-/optimization-time features for the ML models, using simulation for data augmentation, and using regression models to predict job run times for different resource allocations and then using that to determine the optimal allocation.

The platform-specific adaptations are the specific form of the mathematical functions, e.g., power-law function, features used for the ML models and their semantics, the specific ML models, the specific simulators, and the resource allocation unit, e.g., tokens for SCOPE queries and executors for Spark SQL queries.

3 DATA AUGMENTATION

Our goal is to model the relationship between the job run time and the allocated resources (tokens), given the compile-time job characteristics. However, to train such a model, we need the run times for the same job with different token counts so that the relationship between job run time and token count could be learned. Unfortunately, historical data only contains the job run time at one single token count (the one actually used by the job). Here we first discuss challenges along with a couple of alternative approaches, and then we describe our simulation-based technique for augmenting the training dataset.

3.1 Challenges

Augmenting the job telemetry that we use for training is challenging because each of the past jobs ran with a single token count, and their performance over other token counts is unobserved and unknown to us. One option could be to rerun the job with different token counts, leveraging the experimentation capabilities in many data processing platforms, e.g., the job-flighting capabilities in SCOPE that allow re-running production workloads in a pre-production environment [34, 37, 42]. However, such an experimentation is not just time-consuming but also expensive since it requires the availability of sufficient spare tokens to run these additional jobs. Therefore, this approach is not practical for large workloads like the ones we see in SCOPE.

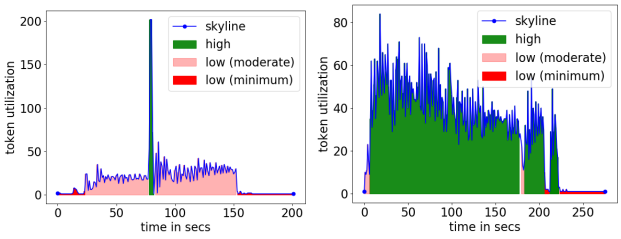
Another approach is to use traditional data augmentation techniques such as the Generative Adversarial Networks (GANs) [19]. Unfortunately, this will not work in our scenario because the GAN models generate new samples based on what we have already observed in the historical data. Instead, we want to produce augmented telemetry at other token counts for each job, i.e., unseen in the historical data. Furthermore, building GAN models to generate different resource consumption skylines is as complex as modeling the run time as a function of token allocation, albeit with limited training data. Overall, we want to keep data augmentation a lightweight process that can scale to large historical workloads and also work for jobs in newer workloads over time.

3.2 AREPAS

Our approach is to use a lightweight skyline simulation technique for data augmentation. Therefore, we introduce ***Area Preserving Allocation Simulator*** (**AREPAS**) to augment the training data in TASQ. AREPAS employs a system-level intuition by assuming that the total amount of work or the area under the resource consumption skyline remains fixed. This makes sense for the same job having the same inputs and the same outputs. Specifically, we discretize the resource consumption skyline at 1 second's granularity, with a 1x1 square in the skyline plot (tokens used over the job's run time) representing 1 token-second, and the total number of squares under the resource consumption skyline representing the total amount of token-seconds used by the job. Our assumption then implies that the total token-seconds will stay constant.

Note that AREPAS simulates performance at the coarse-grained granularity of the entire job and its resource consumption skyline, instead of modeling it for every stage level in the corresponding job execution graph, as proposed in Jockey [18]. This is because simulating at the stage level requires making assumptions about the job scheduler that is much more susceptible to cluster conditions and as a result unpredictable in nature. Instead, AREPAS makes a more simplifying assumption that the total amount of work remains constant. While even this assumption will not always hold, we are still able to obtain run time estimates with reasonable accuracy, as we will show in Section 5.2.

Figure 5 shows two examples of different types of skylines. In these visualizations, we divide the regions under the skylines into different color-coded sections based on the height of the curve in each of those regions, i.e., the utilization of allocated tokens at that point in time. Red indicates near-minimum utilization, pink indicates low utilization, and green indicates moderate-high utilization. We observe that both skylines show potential for cost savings because they spend a portion of the run time in the red/pink section. However, The more peaky job (Figure 5a)



(a) Peaky Skyline **(b) Flatter Skyline**

Figure 5: Resource skylines divided into sections by resource consumption.

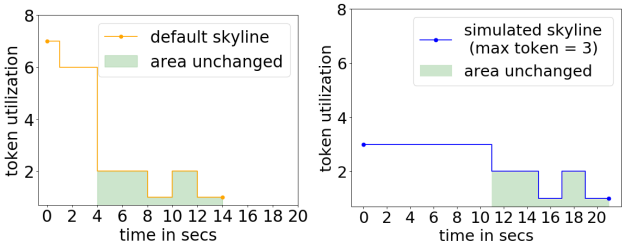


Figure 6: Unchanged section between the Ground truth (left) skyline and simulated (right) skyline.

spends a majority of its run time in the pink/red regions and flatter job (Figure 5b) spends longer time in the green region. We want to simulate the skyline for the above jobs at any token allocation value which is lower than the original token allocation. Additionally, we aim to achieve this using only the above skyline as an input to the simulator. We make the following assumptions.

- The simulated skylines are deterministic, i.e., for any given combination of a job and a token allocation we always get the same skyline. Thus, we do not model any stochastic behavior in the execution environment, e.g., random failures of machines, cluster load at the time of execution, noisy neighbors on same physical machines, scheduler queues, etc.
- Each 1x1 block under the skyline, representing 1 token-seconds each, is independent of each other. So, if a compute process consumes 10 token-seconds over its lifetime, then it will take 1 second and 10 seconds to complete with 10 tokens and 1 token respectively, based on the right-nearest integer approximation.
- Sections of a simulated skyline that are below the allocated resources will have the shape of their skylines unchanged when simulated. (Figure 6)
- For sections of a skyline that go over the allocated resources, the simulator adapts the skyline's shape by reallocating work while keeping the total amount of work done in those sections constant. (Figure 7)

Using the above assumptions, we simulate skylines over different resource allocation for the same job. We divide a resource allocation skyline into sections, where each section is a contiguous chunk of the skyline that is completely under or completely over the new allocation. Figure 6 shows how over-allocated sections (where usage < new allocation) are copied without change to the new skyline. Figure 7 shows how sections of the skyline that are under-allocated (cut-off by new allocation) are immediately added back as a task in front of the part that was cut off. This pushes the rest of the plot forward and as a result it increases the overall job run time. The area that is being added

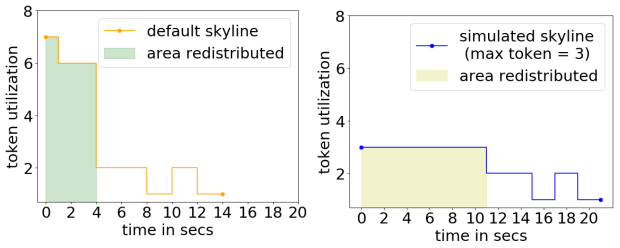


Figure 7: Redistributed section between the Ground truth (left) skyline and simulated (right) skyline. The simulator reallocates work with fewer resources. To preserve area, the reallocated portion in the simulated skyline takes more than twice as long, when allocated a little less than half as many tokens.

is the same as the area that is being cut out, in accordance with the area preservation design choice.

Algorithm 1: Skyline simulation

output : *Ssim*: simulated skyline (*list*) ;
input : *Sog*: original skyline = $[s_0, s_1, s_2, ...s_{runtime}]$
Nt: new allocation threshold to simulate (*int*);

```
1  sectionStartIDs = []
2  for i ← 1 to runtime do
3      if sign(Sog[i]−Nt) ≠ sign(Sog[i − 1]−Nt) then
           sectionStartIDs.append(i);
4  sections = Sog.split(sectionStartIDs):
5  for sec in sections do
6      if sec[0] > Nt then
7          secArea = sum(sec)
8          newSecLength = int(secArea/Mt)
9          repeat newSecLength times
10             Ssim.append(Mt)
11     else
12         Ssim.appendAllElements(sec)
13 return Ssim
```

Algorithm 1 shows the steps for simulating a skyline for a given token count. The input is the original skyline (discretized at 1 second granularity) along with the token count to be used for simulation. The output is the simulated skyline. First, we identify the timestamps where the skyline intersects with the new allocation (Lines 2–3). Then, we use these time stamps to divide the skyline into sections (Line 4). Sections that are above the new allocation are lengthened until they can fit under the new token allocation, while keeping their area under the curve constant (Lines 7–10). Sections that are at or under the new allocation stay unchanged and are copied over as is (Line 12). The new sections are then stacked in front of the others to obtain the newly simulated skyline and the corresponding job run time.

Figure 8 shows the simulation of two kinds of jobs (peaky and flatter) for different token allocations, with the simulated skylines shown in different colors. Note that the peaky jobs show less impact on performance with lower resource allocation than more flatter jobs. This is expected since peaky jobs have deeper valleys and hence more aggressive allocation can shift some of the work to other low activity regions, thus freeing up resources for other jobs.

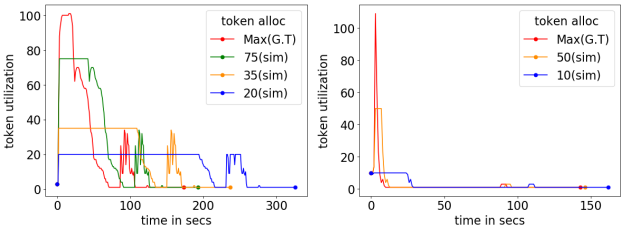


Figure 8: Flatter jobs (left) lose significant performance as soon as the token allocation is decreased, while peaky jobs (right) can tolerate a significant loss of resource allocation before they lose performance. G.T, that is, Ground Truth, denotes the starting skyline for the simulations.

4 PREDICTION MODELS

In this section, we first define the monotonicity constraint of the trend predictions. Then we explain why we prefer a global model compared to fine-grained models. We also discuss how we featurize the job metadata and use ML models to predict job run time as a function of token allocation.

4.1 Monotonicity constraint

The PCC captures the relationship between the job's run time and the number of allocated tokens. Users typically expect to see a monotonically decreasing trend for the run time with increasing token count. While there can be cases in practical situations where the run time increases at high token counts due to parallelism overheads, those are not optimal operating regions for the job, either from a performance or from a cost-efficiency standpoint. However, for SCOPE jobs in particular, re-partitioning of the data or re-optimization of queries do not happen when submitted with different token counts. In this context, we expect to observe monotonic behavior as the dominant common-case scenario. We validate and confirm this hypothesis in Section 5.1.

Our motivations for modeling a monotonic behavior for the PCC, that is, job run times do not increase with increasing token counts, are thus the following.

- It is the common-case scenario for jobs on this platform,
- From a resource allocation perspective, it is not interesting to operate in a region where run time increases with allocated resources, either from the perspective of the user or of the service provider; thus the region of interest is where run times do not increase with token counts, and
- A simple, monotonic PCC is easier to use to explain resource allocation decisions to users.

We consider two alternative approaches for predicting the Performance characteristic curve (PCC).

(1) **Point Prediction**: we predict the run time for a given token amount, then construct the PCC from the collection of predicted points. However, as we will discuss in Section 5.3, the resulting PCC is often not monotonic.
(2) **Trend Prediction**; we fit a monotonic mathematical function to the PCC, then predict the parameters of the function. We discuss the choice of this function below.

From Amdahl's law, the parallelizable portion of a workload has an inverse relationship with run time [8]. However, the steepness of the inverse relationship is not captured by a simple inverse proportionality. To generalize this idea, we characterize the

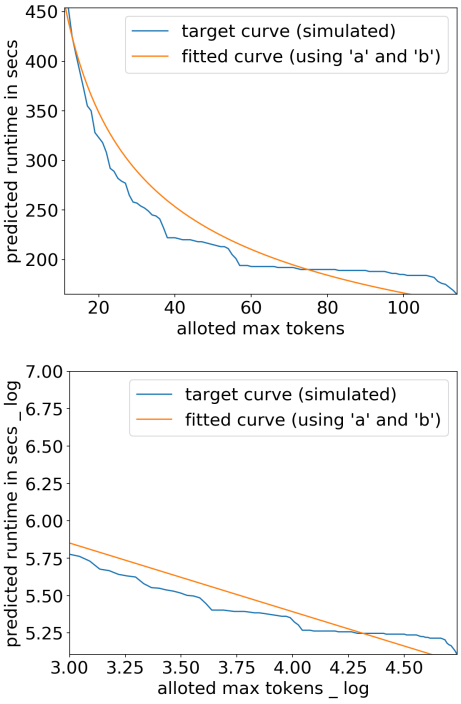


Figure 9: Performance characteristic and the fitted curve in absolute and log-log spaces.

inverse relationship as power-law:

$$Runtime = f\,(A : TokenAllocation) = b \times \mathrm{A}^{a} \qquad (2)$$

Where 'a' and 'b' are the 2 scalar parameters of the curve and whose values depend on the job's characteristics. Amdahl's law can be seen as a special case where 'a' = -1. Thus, the run time versus token relationship would be monotonically increasing if the signs of 'a' and 'b' are consistent and decreasing if the signs of 'a' and 'b' are inconsistent. Since power-law relationships can be represented as a linear curve in the log-log space, we transform the above equation as follows.

$$\log\,(Runtime(A)) = \log\,(b) + a \times \log(A)$$

Figure 9 visualizes the simulated curve and fitted power-law curve in both spaces. We can now fit a straight line (linear regression) through it to estimate the values of 'a' and 'b'.

The above approach for trend prediction has several advantages. First, using just the two parameters for fitting the PCC helps represent the domain knowledge compactly, which not only simplifies the problem but also makes it easier for users to understand the performance-resource trade-off given a PCC. Second, learning a model that can predict the two parameters with good accuracy also guarantees that the monotonicity constraint is satisfied. In contrast, doing a series of point predictions and joining the predictions will not necessarily reach a monotonically non-increasing PCC (as we show in Section 5.3). Furthermore, we can generalize the relationship between job metadata at compile time and performance characteristic to any job, including previously unseen jobs.

A few pitfalls can be expected when using simulated data as the training input. The simulated data is entirely a function of the originally observed points, which violates the assumption about training samples being independent. This makes the simulated data a second-class citizen in the training input and risks teaching the model how to better estimate the AREPAS simulator instead of the run time itself. Fortunately, using our modeling approach allows us to mitigate this to some degree. This is because, one component of the loss function discussed in Section 4.5 only uses ground truth for learning. Thus, the simulator serves as a tool for constraining the model's architecture using inductive bias instead of explicitly guiding it.

We implemented and compared three models: XGBoost [13], feed forward neural networks (NN) and graph neural networks (GNN) in this paper. All these models can be used to make predictions for the power-law curve parameters (Section 4.4), that leads to either monotonically increasing or decreasing PCC. However, we are able to enforce monotonically non-increasing curve by design for NN and GNN (Section 4.5). Below we first discuss the granularity of our models and the featurization, before discussing the models.

4.2 Modeling granularity

We consider two learning granularities:

(1) Global model: a single model for all incoming jobs, both recurring and ad-hoc jobs, and
(2) Fine-grained models: grouping similar jobs together and training separate models for each group, similar as in prior work [34].

Compared to a global model, the fine-grained approach could improve the accuracy by specializing each subgroup's run time versus token count relationship. However, a global model can provide coverage for all jobs, both recurring and ad-hoc jobs, whereas the coverage for the fine-grained approach is limited to recurring jobs seen in the past. Given that token allocation is a difficult parameter for the users to set, we want to predict resource allocation for all incoming jobs and not be restricted to recurring jobs. Therefore, we choose the global model approach in this work, and as we shall show in Section 5.3, the accuracy is quite good as well.

4.3 Featurization

Our models take into account:

(1) job operator characteristics that are available during job compilation and optimization time as summarized in Table 1 (operators are described in more detail in related work [44]), and
(2) the job graph, i.e., a directed acyclic graph (DAG) of the operators as input data.

The featurization and modeling setup differs according to the ML model as we describe in Table 2 and further discuss below.

Modeling the non-linear interactions between a large set of features is difficult with hand-engineering alone. This issue and the presence of labeled data at scale motivated the choice of ML for learning the underlying relationships between run time, resources and features of the submitted job. Each Big Data platform has a different set of features available at compilation time. Once they are grouped into their appropriate variable types and featurization schemes mentioned below, the models outlined in the paper should be able to train and infer from these new features.

Aggregated job-level features. XGBoost and NN models require same input features dimension for each job. As a result, we aggregate the metrics from each operator in the query plan to form an equal length feature vector for each job and feed these models with the aggregated job level features. The continuous and count variables are aggregated by mean, and the categorical

Table 1: Operator level features from SCOPE jobs. Features are grouped by pre-processing and featurization pipelines needed according to variable type.

Variable type	Features encoded in each type
Continuous (float)	Cardinality (estimated: output, leaf input, children input), Average Row Length, Cost (estimated: subtree, operator exclusive, total)
Discrete (integer)	Number of Partition, Number of Partitioning Column, Number of Sort Column
Categorical (one-hot)	35 Physical Operators & 4 Partitioning methods, described in J. Zhou et al. (§5.2, §4.4 [44])

Table 2: Featurization methods and target variables.

Model	Features	Target Variables
XGBoost	Aggregated Job Level	Run time
NN	Aggregated Job Level	PCC Parameters
GNN	Operator Level Graph Representation	PCC Parameters

variables are aggregated by frequency count. The number of operators and stages are included as features as well. For each job, we have a $1 \times P_J$ feature vector, where P_J is the number of aggregated features.

Operator-level features. Unlike XGBoost and NN, GNN is flexible in that the model can take operator level features, as shown in Table 1, directly as input and the input feature dimension could vary by job, depending on how many operators a job has. For each job, the operator-level features are represented as a $N \times P_O$ feature matrix, where N is the number of operators and P_O is the number of operator-level features. Operator-level features avoid the information loss due to aggregation.

Graph representation. We represent the query plan graph structure by an adjacency matrix, and it is obtained from the DAG of the operators. We use this matrix to measure the connectivity of the nodes for the GNN. GNN's native support for learning from graphs motivates its choice as a model.

4.4 Model training

We now describe training three kinds of models, namely, XGBoost, NN and GNN, to learn the parameters of PCC.

XGBoost: Since we cannot use XGBoost [13] to predict the two PCC parameters jointly, we first train XGBoost with Gamma regression trees to predict job run time, then form the PCC with a series of run time predictions. To enable the model to learn a monotonically non-increasing PCC, we perform data augmentation and explore alternate points on both side of the default allocation. For each job, we generate two more observations using the AREPAS simulator at 80% and 60% of observed token amount. For the over-allocated jobs, we observe the peak token allocation and generate two other observations, at 120% and 140% of the peak token allocation, with run time floored at peak allocation. For run time prediction, we consider two approaches to form a PCC as follows.

(1) XGBoost Smoothing Spline (XGBoost SS): The predictions at multiple token counts (within ±40% of the reference, that is, observed token count for the job) are smoothed to form a PCC, and

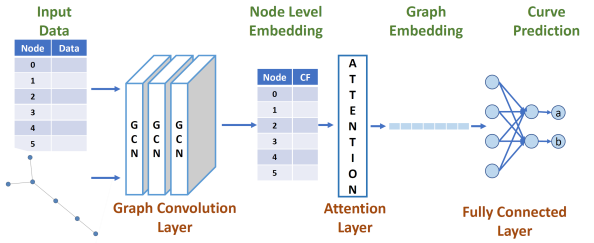


Figure 10: Graph Neural Network architecture.

(2) XGBoost Power-Law (XGBoost PL): The predictions at multiple token counts are used to fit a power-law shaped PCC.

Feed-Forward Fully Connected Neural Networks (NN): The job level aggregated features are fed into a multi-layer fully connected NN to predict the two power-law shaped PCC parameters. The run time at specific token counts can be predicted by plugging the token values into that function. In addition, we can define the peak token count or the optimal token count (e.g., at a specific token count, adding/reducing one token will cause the run time to decrease/increase by p%). This token count can be calculated using the slope and run time predictions based on the predicted PCC function as follows: $\frac{f'(A)}{f(A)} = p\%$

Graph Neural Networks (GNN) We train GNN on operator level features and graph adjacency matrix to predict PCC parameters. We implement a GNN architecture, that is similar to SimGNN [10], which is computationally feasible and efficient, but also takes the node importance into consideration with an effective attention mechanism. Because of the attention mechanism, we can overweigh and focus on the most relevant part of the graph to make accurate run time predictions.

Our GNN consists of three stages, as shown in Figure 10. First, the input data is passed to graph convolution networks (GCN) [28], a neighbor aggregation approach, to obtain the node-level embeddings. Second, the node embeddings are fed into an attention layer, where the attention weight represents the node's similarity to the global context. The global context is a nonlinear transformation of the weighted average of the node embeddings (whose weight is a learnable object), and the graph embedding is the attention weighted sum of the node embeddings. And finally, the job level convolved embeddings are then passed to the multi-layer fully connected NN to predict the PCC parameters.

4.5 Loss function for NN and GNN

For the NN and GNN models, we construct three loss functions using the standard mean absolute error (MAE) loss. We combine several loss components corresponding to both the PCC parameters and run time predictions to form the three loss functions:

LF1: single component loss, MAE of the curve parameters. The parameters are scaled so that neither of the two would dominate the loss function. By scaling these parameters in the training and scaling back in the prediction, the signs of the two predicted curve parameters are guaranteed to be different, which also guarantees the monotonically non-increasing trend between run times and token counts.

LF2: two components loss, a second penalization term of MAE (in percentage) of run time (at the observed token count) is added to regularize the models and improve run time estimates.

LF3: three components loss, a third term of mean absolute difference (in percentage) between the NN/GNN and XGBoost

run time predictions (at the observed token count) is added, with the idea of transfer learning to leverage the learnings from XGBoost (because XGBoost makes good run time point predictions, see Section 5.3). The weights of the components are regarded as hyper-parameters.

5 EVALUATION

We first evaluate the accuracy of the AREPAS simulator. To do this, we gather a ground truth dataset using a small set of jobs, selected based on a job selection process, and re-execute them at different token values using job-flighting capabilities of the platform (Section 5.1). Then, we fit the PCC to obtain the curve parameters. We call this a *flighted dataset* and use it to validate the AREPAS simulator (Section 5.2).

Next, we ran extensive experiments to evaluate the PCC and run time prediction accuracy and to compare the effectiveness of the three ML models, namely XGBoost, NN and GNN, with three different loss functions for NN and GNN. We consider three metrics for model prediction.

(1) *Pattern*, that is, whether or not the predicted PCC is monotonically non-increasing (qualitative metric),
(2) *MAE*, that is, mean absolute error of the curve parameters (quantitative metric), and
(3) *Median absolute error*, in percentage, of run time predictions (quantitative metric).

We train the models with a large production workload of 85K SCOPE jobs after anonymizing identifiable information about the jobs. Both job run time and token utilizations have right-skewed distributions. The job run times range from 33 seconds to 21 hours, with the average and median being 9.5 and 3 minutes respectively. The peak number of tokens used by jobs ranges from 1 to 6,287 tokens, with mean and median of 154 and 54 tokens respectively.

For testing, first we consider a large set of 78K SCOPE jobs, that were submitted a day after the training jobs on the same production cluster. For this dataset, we have the ground truth run time at only one token count for each job. For other token counts, we use estimated run time data obtained from the AREPAS simulator, regarding them as proxy ground truth. We call this a *historical dataset* and evaluate the three metrics of prediction accuracy for the different models (Section 5.3). Second, we further evaluate run time prediction accuracy and token-saving opportunities at the workload level on a *flighted dataset* that we obtain using a similar methodology as before (Section 5.4).

5.1 Flighted dataset for AREPAS validation

In order to construct the flighted dataset for validating AREPAS, we want to obtain run time data for production jobs, where each job is executed with multiple token counts. However, historical data has the job's run time when it was executed at a particular token count. To get the run times at other token counts, we re-executed the jobs at those token counts using the flighting capabilities of the SCOPE processing platform.

However, given that production resources are scarce, we can only re-execute a small fraction of the jobs, each for multiple token counts. So our goal is to construct a (small) subset of jobs that match the population distribution, so that results on this subset could be generalized; and to obtain a subset that contains as many unique jobs as possible. A recent work generates summarized workload sets from the workload population while maximizing coverage and representation [15]. However, in our case, we have to select the subset from a pool of jobs that satisfy certain constraints, e.g., being within certain time frames, having tokens within a particular range, or belonging to a specific virtual cluster. Therefore, we design a simple but effective stratified under-sampling procedure for workload subset selection from the pre-selected pool of jobs. Our procedure includes the following four steps.

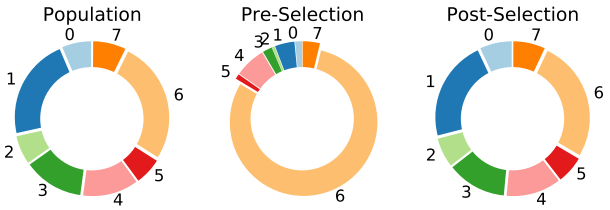


Figure 11: Clusters in the population (left), pre-selection samples (middle) and post-selection samples (right).

1. **Job Filtering**: Filtering the overall workload population based on the constraints (i.e., virtual cluster, token range, time frame) and forming a pre-selected pool of jobs.
2. **Job Clustering**: K-means clustering over the entire population to divide jobs into multiple clusters, and predicting the cluster for each sample in the pre-selected pool of jobs.
3. **Stratified Sampling**: Random under-sampling within each cluster, corresponding to its cluster-size proportion in the population. We further set a threshold value to limit the number of times each type of job can be selected.
4. **Quality Evaluation**: Performing a Kolmogorov-Smirnov (KS) test [5] before and after the job selection. A lower KS statistic after the job selection indicates that the subset distribution is closer to that of the population, compared with the pre-selected pool of jobs.

For validation, we selected 200 jobs using our job selection procedure from the historical data. The population (historical jobs) is split into 8 groups and the cluster size proportion ranges from 5.9% (group 5) to 26.7% (group 6). In the pre-selection samples, however, the majority of (79.9%) jobs fall in group 6 and the smallest sized group only accounts for 0.6% (group 2). After the job subset selection, we are able to create a subset of jobs with cluster-size proportion matching that of the population, as shown in Figure 11. We re-execute each job in the subset with four different token values, at 100%, 80%, 60% and 20% of the original (reference) token count and use the run times as the ground truth in subsequent experiments.

Henceforth, a flight or an execution will refer to a single run of a job with a specific allocation of tokens and the associated time it took to run that once. It is not possible to completely eliminate uncertainty when establishing controls for the flighting experiment. We execute each job 4 times. However, job failures and randomness due to anomalies are always possible. For this reason, each unique flight is run thrice to establish redundancy. We also filter out flights that do not abide by constraints that jobs are expected to follow. The filters are as follows.

(1) Not an isolated flight (that is, we have more than one successful flights of the same job. A minimum of two flights are needed for further analysis),
(2) Max. tokens should not exceed allocation (so, discard errant jobs where the job mistakenly used more than the allocated resources),

(3) Job run time should monotonically decrease with tokens (that is, more compute should not slow the job down).

Therefore, we initially sample a set of non-anomalous jobs that were flighted offline, such that we only analyze those jobs that display non-anomalous behaviors. This added up to 296 flighting instances. They serve as the 'non-anomalous' dataset used for all flighting-related validation henceforth.

To validate the monotonicity assumption in filter (3), we inspected our dataset of 180 uniquely flighted jobs and set a tolerance of less than 10% error to account for environmental/time-of-day conditions. With this limit, we observe that 96% of jobs satisfy the monotonicity constraint. For the remaining 4%, i.e., 8 of 180 jobs that violate the constraint, the average slowdown due to more resources was 14% compared to the minimum run time for the job.

5.2 AREPAS Validation Results

We now validate the area-preserving assumption in AREPAS, i.e., "*The total token-secs or area under the resource consumption curve stays constant*". To do this, we run each job with 4 different token counts and compare the area under the resource consumption curve (skyline) for each pair of executions. Four executions per job provide ${}^4C_2 = 6$ execution pairs. We consider an execution pair to uphold the area conservation assumption if the percentage difference in area is under a specified tolerance range. As we relax our tolerance by increasing the range, a greater proportion of execution pairs will be considered matches. Figure 12 (*top*) shows the CDF over all tolerance ranges. We observe that if we set the tolerance range to 30%, then 65% of all execution pairs match with each other. When viewed on a granular job-by-job basis, if an execution does not match with the rest of its executions, we refer to it as an outlier. Figure 12 (*bottom*) shows the prevalence of outliers over the 4 executions of each job. 83% of all jobs have 1 or fewer outliers over their 4 executions. Having evaluated our assumption at both the aggregate and granular level, we find AREPAS's core assumption of token-seconds staying constant to be reasonable.

Figure 12 compares the relationship between the percentage tolerance and the percentage matches of job execution pairs. Let us set the tolerance at 10%, and compare 2 flights of the same job at different tokens. As long as the areas under the skylines for both flights are within 10% of each other, we will consider it a match. In Figure 12, we see that at 10% tolerance, 50% of all flights that were compared were considered matches. Along the same vein, we also see that almost all comparisons lie within a tolerance of 100%. This means that in our test flights, the area under the skyline curve for one flight of a job was almost never twice as much as another flight of the same job at a different token value.

We now validate whether the simulations generated by AREPAS match those of the re-executed job instances. To focus on deterministic patterns, we discard jobs with any anomalous behavior as outlined in Section 5.1.

Alongside the non-anomalous dataset, we also analyze a smaller (fully-matched) subset of jobs which exhibit zero outliers on the green line as seen in Figure 12 (*bottom*). These are jobs where token-secs match for all four executions. They represent 38% of the original sample. We report Mean Average Percent Error (MeanAPE) and Median Average Percent Error (MedianAPE) for both the non-anomalous subset and the fully-matched subset in Table 3. We observe, that AREPAS's results match the re-executed

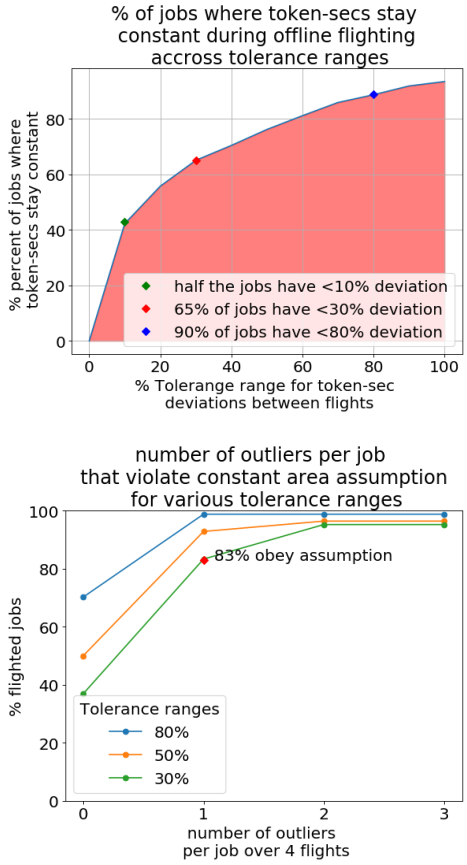


Figure 12: Validating constant resource allocation area assumption: 83% jobs uphold the assumption within a tolerance of up to 30% and with 1 or fewer outliers.

Table 3: AREPAS error compared to ground truth.

Job Groups	N Executions	MedianAPE	MeanAPE
Non-anomalous subset	296	9%	14%
Fully-matched subset	97	22%	25%

run times quite closely, with the worst-case error being under 50% and 30% for the non-anomalous and fully-matched subsets respectively. This can be seen in Figure 13 at the 100 percentile point of both plots. The spikes on Figure 13's percent histogram indicate AREPAS's median average percent error on each subset.

5.3 Model accuracy on historical dataset

We now validate the model predictions over the historical dataset.

Table 4–6 shows the performance comparison of the three models. For trend prediction, neither of the XGBoost models (XGBoost SS, XGBoost PL) can guarantee a monotonically non-increasing PCC even after data augmentation. For XGBoost SS, only 41% of the jobs have non-increasing PCC (within a region of ±40% of the reference token count for the job i.e., the allocated token count for the actual run), and around 60% the predicted PCCs have at least one region of increase. For XGBoost PL, around 73% of the jobs have predicted PCC non-increasing (the two predicted curve parameters have different signs), and 27% of the jobs have the predicted PCC monotonically increasing (predicted

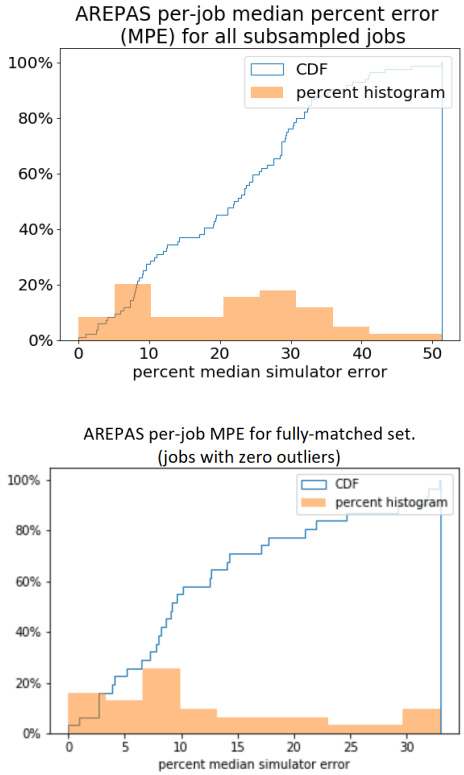


Figure 13: AREPAS accuracy against the ground truth: median error for jobs with non-anomalous behavior is 9.2%.

curve parameters have the same sign), i.e., run time increases with token counts. This is because point predictions for XGBoost do not necessarily decrease with the increase of tokens. For NN and GNN, a monotonically decreasing PCC is fitted and guaranteed, because the curve parameters are scaled and treated directly as target variables. GNN has a smaller error (MAE) of curve parameters (0.071–0.077), compared to NN (0.083–0.090). The MAE of curve parameters for XGBoost PL is much higher (0.232), around 3× that of NN and GNN.

For run time prediction at the reference token count for each job, XGBoost models have smaller errors since they model the run time directly and aim to minimize the run time error only. Half of the jobs have run time prediction error of 13% or less. The median absolute error (Median AE) of run time prediction for NN and GNN are similar, but larger than that for XGBoost. The run time prediction accuracy also depends on the loss function.

NN and GNN have the largest errors with LF1 because it has a single target to minimize the error of curve parameters. For LF2, the penalization term is added and as a result, the models aim to minimize the run time prediction error as well. The curve parameter loss and run time related loss are balanced by applying weights. We tuned the penalization weights, so that the MAE of the curve parameters in LF2 is close to that of LF1. Adding the penalization terms substantially improves the run time prediction for NN and GNN (31% Median AE for LF1 vs. 20–22% for LF2) without sacrificing the accuracy of curve parameters prediction. When comparing LF2 with LF3, the NN/GNN model errors do not differ significantly. So, adding another penalization term to take advantage of XGBoost's learning is redundant.

Among the models that we studied, XGBoost SS makes no assumption of the shape of the PCC. XGBoost PL, NN, and GNN

Table 4: Results for 1st form of the loss function (LF1).

Model	Pattern (Non-Increase)	MAE (Curve Params)	Median AE (Run Time)
XGBoost SS	41%	NA	13%
XGBoost PL	73%	0.232	13%
NN	100%	0.086	31%
GNN	100%	0.071	31%

Table 5: Results for 2nd form of the loss function (LF2).

Model	Pattern (Non-Increase)	MAE (Curve Params)	Median AE (Run Time)
XGBoost SS	41%	NA	13%
XGBoost PL	73%	0.232	13%
NN	100%	0.090	22%
GNN	100%	0.071	20%

Table 6: Results for 3rd form of the loss function (LF3).

Model	Pattern (Non-Increase)	MAE (Curve Params)	Median AE (Run Time)
XGBoost SS	41%	NA	13%
XGBoost PL	73%	0.232	13%
NN	100%	0.083	22%
GNN	100%	0.077	21%

assume a power-law shaped PCC. However, the XGBoost models do not guarantee a non-increasing pattern for the PCC. We obtain the non-increasing curve from XGBoost SS for less than half of the jobs within a local range (±40%) of the reference point. For XGBoost PL, the predicted run time has a monotonically increasing relationship with token count for 27% jobs. This could cause confusion to users as an increase in job run time with more tokens is not expected. Also, the XGBoost model performance outside the local range of the reference point is not expected to perform as well. Thus, the construction of the range as well as the prediction of the reference point are needed.

In contrast, the NN and GNN models with LF2 are accurate for both trend and point predictions. We prefer NN/GNN to XGBoost. They aim to learn the shape of the PCC from AREPAS and predict run times using the predicted power-law curve that is monotonically decreasing for the whole range. Both models make assumptions of the PCC shape (specified by two parameters).

Table 7: Parameter counts, training and inference times.

Model	Number of Parameters	Time(s): Training Per epoch	Time(s): Inference Per 10,000 jobs
NN	2,216	2	**0.09**
GNN	19,210	913	78

As shown in Table 7, NN is more lightweight than the GNN model (9× fewer parameters), is less computationally intensive (450× faster training, 900× faster predictions), and does not require the job graph data. The GNN model shows marginally better performance than NN, but requires job graph data and more computational capacity. Thus, while both XGBoost and GNN models excel in certain areas, they fall short towards some

Table 8: Results on the flighted dataset.

Model	Pattern (Non-Increase)	MAE (Curve Params)	Median AE (Run Time)
XGBoost SS	32%	NA	53%
XGBoost PL	93%	0.202	52%
NN	100%	0.163	39%
GNN	100%	0.168	**33%**

critical constraints with regards to expected trends and speed respectively. On the other hand, the NN model strikes a balance by sufficiently satisfying constraints towards accuracy, required trends and speed.

5.4 Model accuracy on flighted dataset

To evaluate model prediction accuracy against ground truth data on multiple token counts for the same job, and since it is infeasible to flight a large number of jobs, we obtain a flighted dataset by selecting a representative sample of jobs, as discussed in Section 5.1, from the test set and running each job with multiple token counts. The successfully-flighted dataset comprised of 31 jobs, 97 runs, and 67 unique token counts, with 2–4 token selections per job.

Table 8 summarizes the errors for XGBoost SS, XGBoost PL, NN and GNN with LF2 on this flighted dataset. The errors for XGBoost are the largest, followed by NN and GNN. For point prediction, the Median AE of run time predictions for XGBoost is 4.1× that for the full historical jobs augmented with AREPAS (53% vs 13%; also see Table 5), while for NN and GNN it is 1.8× (39% vs 22%) and ∼1.7× (33% vs 20%) respectively. For trend prediction, the MAE of curve parameters for NN and GNN are 1.8× and 2.4× that of the full historical jobs, while for XGBoost PL it is smaller. NN and GNN have smaller error on both point and trend prediction, compared with XGBoost. Predicted PCCs for NN and GNN have less than 17% error in curve parameter estimation and all have the monotonically non-increasing pattern, while in the flighted dataset 28 of the 31 jobs have monotonically non-increasing PCCs (within the 10% tolerance limit mentioned in Section 5.1). GNN has the smallest Median AE (33%) for run time predictions amongst the models.

To evaluate the token-savings opportunities at the workload level against predicted and actual performance impacts, we considered the following workloads from the flighted dataset. For each workload, we consider the baseline as using the largest flighted token count for each job, for every run of that job in the workload.

- W1: this is the entire flighted dataset with all 97 runs of the 31 jobs and using the flighted token counts for each run. W1 uses a total of 6.7K tokens. Baseline B1 uses 8.6K tokens.
- W2: this consists of 31 runs, one for each job in the flighted dataset, using the second-largest flighted token count for that job. W2 uses a total of 2.4K tokens. Baseline B2 uses 3K tokens.

Thus, W1 and W2 respectively save 23% and 20% of total tokens compared to the baselines B1 and B2. Considering total run times for the workload, W1 and W2 incur slowdowns ($slowdown = (newtime/baselinetime) - 1$) of 18% and 8% compared to B1 and B2, thereby showing trade-offs between token savings and workload performance. The GNN model predicts a workload-level slowdown of 8% and 5% for W1 and W2, which is reasonably close compared to the actual slowdowns.

6 RELATED WORK

We first mention a few other contexts where inductive bias based on domain knowledge has been useful. Then we discuss prior approaches for automatically allocating the optimal amount of resources, and how TASQ extends the state of the art. Following that we describe a couple of other simulation techniques and their limitations.

6.1 The rise of ML and the role of inductive bias

The rise of big data has been accompanied by the meteoric rise of the field of Machine Learning (ML), which has made significant advances by exploiting the exponential increase in data and computational resources. ML methods have been employed to great success in various domains by finding creative ways of injecting domain knowledge as constraints in the ML architecture, also referred to as inductive bias [11]. We enumerate a few such examples below.

Convolutional neural networks [29, 30] used sliding window filters to learn spatial relationships inherent in visual structures. LSTM Networks [23] and Transformers [39] exploit directionality, memory, and context inherent in language to obtain state of the art performance for natural language use-cases. Graph Neural Networks [20, 33] are slowly gaining momentum as promising tools for understanding data structured in the form of a graph.

In each case, building machine learning models around inductive biases that have been derived from domain knowledge of the problem space has yielded successful outcomes.

6.2 Optimal resource allocation

Efficient resource allocation has received a lot of attention in recent years [7, 14, 16, 18, 26, 27, 32, 34]. Amazon recently announced about Predictive Scaling in EC2 where ML models are used to predict traffic patterns and create a scaling plan [6].

For SCOPE, Jockey [18] and PerfOrator [32] study efficient resource allocation using non-ML approaches. Morpheus [26] considers past resource-usage skylines and uses linear programming for their job resource model. However, it does not model critical relationships, e.g., those between the amount of data and run time.

Autotoken [34] groups recurring SCOPE jobs by signatures and then trains individual off-the-shelf models for each group to capture relationships between data size as well as compile-time estimates of job metadata and a group's peak allocation. AutoToken does not predict resource allocations for ad-hoc jobs. For SCOPE, between 40–60% [34] of the query jobs are new and do not have previous runs to act as a reference. Also, AutoToken does not do time prediction, and thus is unable to answer what-if questions regarding impact of sub-peak token allocations on job run time. AutoToken only considers aggregate job-level characteristics, not the shape or individual operators of the job plan. TASQ, in contrast, constraints the model to the expected performance characteristics to learn a PCC and can do trend predictions using that. It also uses a novel simulator to cheaply yet accurately augment the training data.

A prior work [9] adapts resources to the skyline progressively by estimating peaks in the job's remaining lifetime and releasing excess resources. But it cannot reclaim resources more aggressively. It also requires deep integration and constant communication with the job scheduler during the job's run time and

cannot provide estimates or insights into the job's execution at compilation time.

Wang et al. [41] use two-step classification with black-box ML models to tune Spark configuration parameters. Fan et al. [16, 17] use ML models on compile-time estimates of query characteristics to predict run times of SQL queries for different Degrees Of Parallelism (DOP) in order to select the optimal DOP. In contrast, in addition to point prediction, TASQ also uses novel techniques to do trend prediction and uses a novel simulator to augment training data.

Venkataraman et al. [40] try to capture this relationship by fighting an observed job for a variety of resources, but at a much smaller scale of under-sampled data and lower compute, and then learn generalized linear models. Their predictions use up on average, 4% of the job's total run time, and needs to be run for every job that is submitted, implying that deploying such a feature would add a blanket 4% overhead on average to all jobs submitted to Cosmos.

Zhang et al. [43] built a similar performance modeling tool for MapReduce, where they learn linear relationships between data and each of the 6 processes that encompass a standard MapReduce pipeline. However, the assumption of linear scaling of run time with data does not always hold. In SCOPE the number of operators are much larger (52) and further extensive to any number using UDOs. They also do not consider the nature of the data itself, in terms of rows or data format. SCOPE job graphs are almost always complex DAGs which prevents us from making the same assumptions.

Elastisizer and Starfish [21, 22] studied the problem of automatically predicting optimal cluster sizes for MapReduce jobs. In their work, the cost statistics and dataflow properties for jobs are modeled differently, with profiling and M5 tree models used to model the former and white-box/analytical models used to model the latter. The model prediction results are used by a task scheduler simulator to simulate the execution of the job on the target cluster. A separate simulation/estimation is needed for each candidate cluster size (point prediction). In contrast, TASQ does trend prediction for the PCC in addition to point prediction, and uses simulation only for training data augmentation. While TASQ does not use analytical models, it uses a simple power-law formulation for the PCC to make resource allocation decisions easily explainable to users.

6.3 Simulators for SCOPE

Within the optimal resource allocation domain, many different simulators have been proposed. These methods attempt to simulate a job's execution time for previously unobserved resource allocations. For SCOPE, Ferguson et al.[18] have introduced two different job simulators as a part of Jockey. We refer to them as the Jockey simulator and the Amdahl's law[8] simulator.

The Jockey simulator looks at previous runs of the same job, and records stage-level performance statistics aggregated over all historic runs of that job. These aggregated statistics include distributions of task run times, initialization latency, and the probabilities of single and multiple task failures. A key statistic that is not captured is input size variation. The simulator then consumes the job's Algebra, allowing to simulate each stage of the job based on statistics gleaned from previous runs. The simulator's parameters can be prohibitively expensive to compute online. As a mitigation, exhaustive simulations are run offline for different resource allocations a and progress points p in the job's life cycle to estimate a distribution $C(p, a)$ which can then be used online for no extra cost.

Similar to the Jockey simulator, the Amdahl's law simulator operates at a stage-level granularity and has the same limitations. This simulator divides each stage into a serial(S) and parallel(P) part. Here, S is the critical path for each stage and (N) represents tokens, which serve as a proxy for the resources allocated to the job. The simulator states that the run time T can be formalized as: $T = S + P/N$. For each stage, S and P are computed as aggregated statistics obtained from prior runs of the job. When used for predicting run time for a job at compile time, the Jockey simulator performs identically to the Amdahl's law simulator.

Slow online run times and inability to extend to fresh jobs or capture how simulations may vary with data size are the main shortcomings of the Jockey and Amdahl's law simulators. Both simulators encode information that is rich in domain knowledge. Being able to inject this information into an ML architecture as inductive bias might be a promising avenue to explore. However, since both simulators are represented by a large number of stage-level parameters, ML models are unable to sufficiently constrain these rich representations to learn anything of value.

7 CONCLUSION

This paper pushes the envelope for resource optimization in big data analytics. In contrast to peak resource allocation in prior works, we target optimal resource allocation that trades small to minimal loss in performance for more aggressive allocation to improve the overall efficiency. We presented TASQ, an end-to-end learning approach to capture the performance characteristics curves (PCC) of analytical jobs as a function of resource allocation. TASQ is flexible in that it enables users to choose their own scheduling policy based on the PCC, but is also able to predict the optimal token counts automatically, without user inputs, based on a preset threshold. We model the PCC as a power-law curve with monotonicity constraints, to be in line with user expectations, and compared three kinds of models with multiple loss functions. Our evaluations on production workloads showed that PCCs can be well-predicted, with median run time prediction errors of 39% or less at individual token counts.

One of the challenges in training ML models to learn the relationship between job run time and resource allocation is obtaining sufficient training data. Historical datasets of production workloads usually include runs of jobs at a given token count for each job, but we also needed run times of the job at other token counts. While replay/re-execution of jobs at other token counts is possible, this is costly and time consuming in addition to needing platform support for the capability. To address this challenge, we introduced the AREPAS simulator that can be used to efficiently augment limited ground truth data with synthetically generated skylines at different token allocations for jobs that have run in the past. Our results show that run time estimates from AREPAS are close to actual run times, with a worst-case estimation error of less than 50%.

ACKNOWLEDGMENT

We thank Vishal Rohra, Anubha Srivastava, Yi Zhu, Hiren Patel, Shi Qiao, Marc Friedman, Carlo Curino, and other members from the MAIDAP, SCOPE, and GSL teams who have contributed, shared feedback, or provided useful directions throughout the TASQ project, and the anonymous reviewers for their valuable feedback on this paper.

REFERENCES

[1] 2022. *Amazon Athena.* https://aws.amazon.com/athena/

[2] 2022. *Azure Data Lake Storage.* https://azure.microsoft.com/en-us/services/storage/data-lake-storage

[3] 2022. *Azure Kubernetes Service.* https://azure.microsoft.com/en-us/services/kubernetes-service

[4] 2022. *Azure Machine Learning.* https://ml.azure.com

[5] 2022. *Kolmogorov–Smirnov test.* https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test

[6] 2022. *New - Predictive Scaling for EC2, Powered by Machine Learning.* https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/

[7] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 469–482.

[8] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. 483–485.

[9] Malay Bag, Alekh Jindal, and Hiren Patel. 2020. Towards Plan-aware Resource Allocation in Serverless Query Processing. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association. https://www.usenix.org/conference/hotcloud20/presentation/bag

[10] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. SimGNN: A Neural Network Approach to Fast Graph Similarity Computation. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining* (Melbourne VIC, Australia) *(WSDM '19)*. Association for Computing Machinery, New York, NY, USA, 384–392. https://doi.org/10.1145/3289600.3290967

[11] Jonathan Baxter. 2000. A model of inductive bias learning. *Journal of Artificial Intelligence Research* 12 (2000), 149–198.

[12] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1265–1276.

[13] Tianqi Chen and Carlos Guestrin. 2016. XGBoost. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Aug 2016). https://doi.org/10.1145/2939672.2939785

[14] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. 2018. Stratus: Cost-Aware Container Scheduling in the Public Cloud. In *SoCC '18* (Carlsbad, CA, USA). Association for Computing Machinery, New York, NY, USA, 121–134. https://doi.org/10.1145/3267809.3267819

[15] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeff Naughton, and Stratis Viglas. 2020. DIAMetrics: Benchmarking Query Engines at Scale. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 3285–3298. https://doi.org/10.14778/3415478.3415551

[16] Zhiwei Fan, Rathijit Sen, Paraschos Koutris, and Aws Albarghouthi. 2020. Automated tuning of query degree of parallelism via machine learning. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.

[17] Zhiwei Fan, Rathijit Sen, Paraschos Koutris, and Aws Albarghouthi. 2020. A Comparative Exploration of ML Techniques for Tuning Query Degree of Parallelism. *arXiv preprint arXiv:2005.08439* (2020).

[18] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*. 99–112.

[19] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'14)*. MIT Press, Cambridge, MA, USA, 2672–2680.

[20] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, Vol. 2. IEEE, 729–734.

[21] Herodotos Herodotou, Fei Dong, and Shivnath Babu. 2011. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-Intensive Analytics *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 18, 14 pages. https://doi.org/10.1145/2038916.2038934

[22] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 261–272. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper36.pdf

[23] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[24] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload Optimization for Cloud Query Engines. In *SoCC '19* (Santa Cruz, CA, USA). Association for Computing Machinery, New York, NY, USA, 416–427. https://doi.org/10.1145/3357223.3362726

[25] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[26] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards automated SLOs for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 117–134.

[27] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang. 2016. Hadoop Performance Modeling for Job Estimation and Resource Provisioning. *IEEE Transactions on Parallel and Distributed Systems* 27, 2 (2016), 441–454.

[28] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105.

[30] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[31] Anish Pimpley, Shuo Li, Anubha Srivastava, Vishal Rohra, Yi Zhu, Soundararajan Srinivasan, Alekh Jindal, Hiren Patel, Shi Qiao, and Rathijit Sen. 2021. Optimal Resource Allocation for Serverless Queries. *arXiv preprint arXiv:2107.08594* (2021).

[32] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. 2016. PerfOrator: Eloquent Performance Models for Resource Optimization. In *SoCC '16* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 415–427. https://doi.org/10.1145/2987550.2987566

[33] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2008), 61–80.

[34] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. 2020. AutoToken: Predicting Peak Parallelism for Big Data Analytics at Microsoft. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3326–3339.

[35] Rathijit Sen, Abhishek Roy, and Alekh Jindal. 2021. Predictive Price-Performance Optimization for Serverless Query Processing. *arXiv preprint arXiv:2112.08572* (2021).

[36] Rathijit Sen, Abhishek Roy, Alekh Jindal, Rui Fang, Jeff Zheng, Xiaolei Liu, and Ruiping Li. 2021. AutoExecutor: Predictive Parallelism for Spark SQL Queries. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2855–2858.

[37] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *SIGMOD '20* (Portland, OR, USA). Association for Computing Machinery, New York, NY, USA, 99–113. https://doi.org/10.1145/3318464.3380584

[38] Jordan Tigani and Siddartha Naidu. 2014. *Google BigQuery Analytics*. John Wiley & Sons.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5998–6008.

[40] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *USENIX (NSDI 16)*. Santa Clara, CA, 363–378. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[41] G. Wang, J. Xu, and B. He. 2016. A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 586–593.

[42] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *Proceedings of the VLDB Endowment* 12, 3 (2018), 210–222.

[43] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. 2013. Benchmarking Approach for Designing a Mapreduce Performance Model. In *ICPE '13* (Prague, Czech Republic). Association for Computing Machinery, New York, NY, USA, 253–258. https://doi.org/10.1145/2479871.2479906

[44] Jingren Zhou, Nicolas Bruno, Ming chuan Wu, Per ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *The VLDB Journal* 21, 5 (2012), 611–636.