Predictive Price-Performance Optimization for Serverless Query Processing

Rathijit Sen
Microsoft
rathijit.sen@microsoft.com

Abhishek Roy*
Keebo
aroy@keebo.ai

Alekh Jindal*
Keebo
alekh@keebo.ai

ABSTRACT

We present an efficient, parametric modeling framework for predictive resource allocations, focusing on the amount of computational resources, that can optimize for a range of price-performance objectives for data analytics in serverless query processing settings. We discuss and evaluate in depth how our system, *AutoExecutor*, can use this framework to automatically select near-optimal executor and core counts for Spark SQL queries running on Azure Synapse.

Our techniques improve upon Spark's in-built, reactive, dynamic executor allocation capabilities by substantially reducing the total executors allocated and executor occupancy while running queries, thereby freeing up executors that can potentially be used by other concurrent queries or in reducing the overall cluster provisioning needs. In contrast with post-execution analysis tools such as Sparklens, we predict resource allocations for queries before executing them and can also account for changes in input data sizes for predicting the desired allocations.

1 INTRODUCTION

Modern clouds have democratized data analytics such that users can easily sign up and get access to the most sophisticated analytics platforms in the cloud. As a result, most of the complexities in owning and operating these data analytics platforms have been taken care of by the cloud providers, and they offer a way simpler pricing model based on the amount of resources actually used. The newer serverless query processing models, such as those in AWS Athena [13], Google BigQuery [17], Synapse Spark [6], and Synapse SQL [9], have further alleviated the need for users to provision any dedicated resources and instead these new serverless approaches automatically allocate resources on a per-query level. These trends have also led to data analytics becoming extremely resource intensive in modern clouds due to the massive amount of data they consume and the complex processing they apply over it. As a result, it is important for enterprises to manage their total cost of operations (TCO) by reducing their resource consumption and doing more analytics with less resources.

Current approaches for optimizing cloud resources are reactive in nature. For instance, several efforts have considered recommending SKU[1] or other resource recommendations based on past usage patterns, e.g., SKU recommendations in SQL Server [11]. Others such as Sparklens [5] analyze the performance of a previously executed Spark query to suggest better resource configurations, e.g., number of Spark executors (worker processes) to use. Still other automatic approaches include detecting idle cycles to pause or resume the system as in Azure SQL [8], auto-scaling the resources either at the cluster or pool level as in Synapse Spark [26], RedShift [28], Snowflake [20], etc., or at the query level as in Spark [10] or even Cosmos [30] based on the availability of spare resources. Unfortunately, these reactive approaches take several minutes to react [26] and many of the optimization opportunities may already be missed. Additionally, reactively adjusting resources during the course of a query execution could even lead to expensive changes in the query plan [55]. Therefore, apart from the reactive approaches, we also need predictive resource allocation to provide a good starting point.

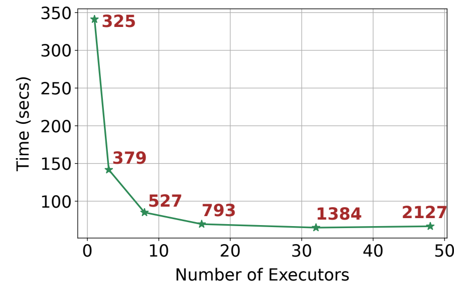


Figure 1: Average application run time and area under the executor allocation skyline (data labels) for an example TPC-DS query, when run with different executor counts.

Predictively allocating resources to an analytics query is challenging since it is non-trivial to map a query to its resource needs. In fact, it is well known that even expert users cannot correctly estimate the resources needed for a given query [41]. Furthermore, changing resources for a query impacts its performance. To illustrate, Figure 1 shows the performance of an example TPC-DS query implemented in Spark SQL when using different number of Spark executors, the unit of resources available to queries in Spark. The green curve in Figure 1 shows that the performance improves (lower runtime) initially as more executors are added, however it plateaus later on. The data labels in red show the total executor occupancy, measured as the sum of durations (seconds) over which each executor was allocated to the query, across the entire query execution. We see that even when the performance plateaus, the resource consumption continues to increase with more executors. Thus, we have an interesting price-performance optimization at hand.

In this paper, we study predictive price-performance optimization in serverless query processing setting, i.e., resources are allocated and users are charged at the query level. We build on top of our prior work on the relationship between query performance and resources in Hive and Spark [55], predictive degree of parallelism in SQL Server [34], peak [52], adaptive [29] and optimal [47, 48] resource allocation in SCOPE [32] jobs, and present an end-to-end framework for predictive price-performance optimization at the query level.

Specifically, we make the following key contributions.

(1) We present a framework for predicting optimal resource allocations using simple and explainable parametric models

---

*Work done while at Microsoft.
[1] Stock Keeping Unit

© 2023 Copyright held by the owner/author(s). Published in Proceedings of the 26th International Conference on Extending Database Technology (EDBT), 28th March-31st March, 2023, ISBN 978-3-89318-088-2 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

[Section 3]. We extend prior work in this space by explicitly modeling for performance saturation, the absence of which can cause allocations to severely overshoot desired levels when optimizing for cost with a maximum allowable slowdown [Section 5].
(2) We show for the first time, the distributions of executor and core counts used by hundreds of thousands of Spark queries on a commercial platform. By default, these parameter values are small for a vast majority of queries, which may be a suboptimal choice from a performance maximization standpoint. This suggests the opportunity for substantial performance improvements by configuring these parameters according to the query characteristics [Section 2].
(3) We show how our system, *AutoExecutor*, integrates predictive allocation within the Spark query optimizer and leverages existing mechanisms for reactive deallocation to free up idle resources. We discuss, in depth, various technical challenges to achieve this and how our solution overcomes these challenges to create an efficient end-to-end system [Section 4].
(4) We compare the benefits of our system, *AutoExecutor*, with two state-of-the-art techniques for Spark: Sparklens [5] and Dynamic Allocation [10]. We overcome Sparklens' limitations of not being able to predict optimal resources for ad-hoc queries before executing them or handling changes in input data sizes, while achieving similar prediction accuracy. We also demonstrate substantial cost savings (median 37.5% savings in per-query executor allocations, median 20.3% savings in per-query executor occupancy and overall 16.1% savings for the workload) through *AutoExecutor* compared with Dynamic Allocation, and to the best of our knowledge, this is the first work to demonstrate substantial savings with predictive allocation over state-of-the-art reactive allocation in Spark [Section 5].

We recently demonstrated our system design and concept in a 4-page demo paper [53]. In this current paper we present the modeling framework and architecture in more detail, evaluate additional models, and analyze feature importance and sensitivity to input data size changes. We also present a characterization of resources used by Spark queries in production workloads, details of Spark optimizer extensions that combine both the predictive and the reactive approaches, and show substantial cost savings compared to Dynamic Allocation.

2 QUERY RESOURCE ALLOCATION

In this section, we discuss various query-level resource allocation approaches. We focus our discussion on Spark query processing engine in this paper. We consider the executor count, i.e., the number of worker processes, available to each Spark SQL query as a unit of resource since it is a crucial factor in both query performance and cost. Later we discuss how we can extend this to select the total number of cores as well.

The query processing cost is determined by the resource allocated to execute it. In this work, we focus on the computational resources. Let n_s denote the number of executors allocated to a query at time s during its execution. We are interested in the following two metrics.

(1) The maximum executor allocation, $n = max(n_s)$, that impacts both query performance and total provisioning needs. The query performance is inversely proportional to its total run time, which we denote by $t(n)$.
(2) The total executor occupancy, which is the sum of time intervals during which each executor is allocated to the query. If we consider the resource skyline [41, 50], that is, a timeline plot of n_s vs s over the duration of the query execution, then the total executor occupancy is the Area Under the (skyline) Curve which we denote by AUC. It reflects the total resource reservation cost and can be calculated as $AUC = \int_s n_s ds$.

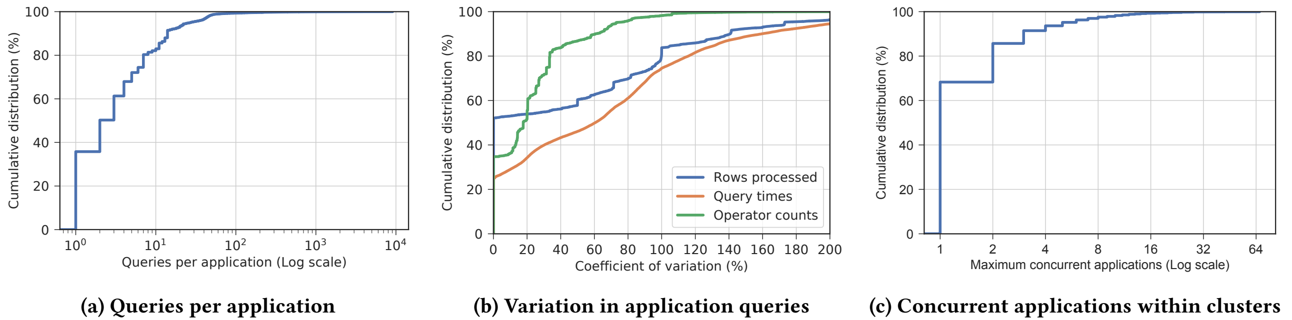
As Figure 1 shows, both $t(n)$ and AUC are strongly influenced by n. In this work, we do not directly predict AUC or optimize for a specific AUC target, but focus on modeling the relationship of $t(n)$ as a function of n. We refer to this as our Price-Performance Model (PPM). Section 5.4 shows how our techniques help to reduce AUC.

2.1 Why Per-query Resource Allocation?

The first question is why does per-query resource allocation matter. We do statistical analyses on non-identifying telemetry information for a large subset of daily production Spark workloads at Microsoft consisting of more than 90K applications and 840K queries across more than 3.2K clusters. Figure 2a shows the distribution of the number of queries per Spark application. We can see that more than 60% of the applications have more than one query and so all of them need to be considered if we were to allocate resources at the application level. Furthermore, Figure 2b shows that the variation of queries within each of the Spark applications. We see that queries in half of the applications exhibit a coefficient of variation [18] of 20% or more in their operator counts, 40% or more in the number of input rows processed, and 60% or more in their query execution times. As a result, queries within an application are quite different and are expected to have different resource requirements that might be hard to aggregate at the application level. Finally, Figure 2c shows the number of concurrent Spark applications in a cluster at any given point, and we see that around 70% of the applications do not share compute resources with other applications in the same cluster. The contention for shared resources is further reduced by the dedicated compute pools and disaggregated storage service [19] in Azure Synapse [6]. This also makes *AutoExecutor* applicable to other systems that leverage the cloud's ability to independently scale storage and compute resources like Snowflake [24]. Thus in cloud systems, it is more practical to allocate resources efficiently at per-query level within each Spark application.

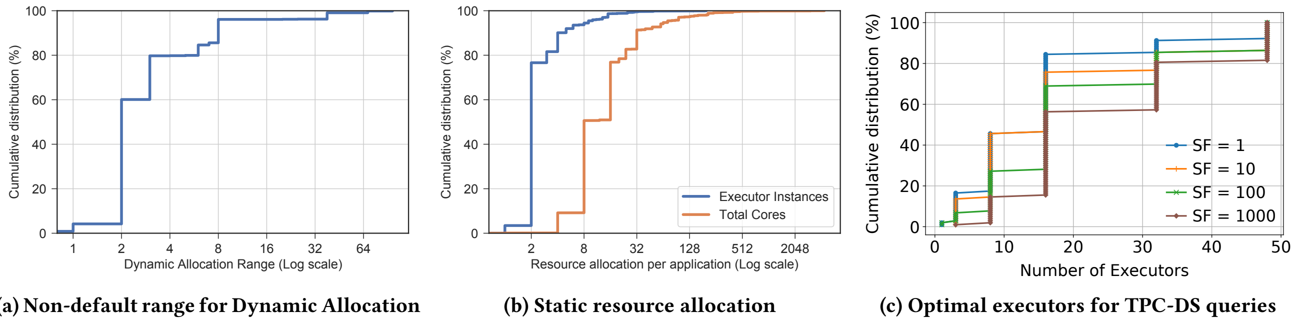
2.2 The Default Behavior

Our production Spark workloads show that 59% of the Spark applications have Dynamic Allocation [10] enabled, which is a reactive approach to adjust the executor count based on the pending requests (more on Dynamic Allocation below). Interestingly, 97% of these applications with Dynamic Allocation enabled also have the default minimum and maximum executor threshold set, which are 0 and $2^{31} - 1$ respectively. For the remaining 3% applications users set their own values, and Figure 3a shows the distribution of the executor count range in those applications. We can see that almost 60% of these applications have a range of just 2, while the remaining could have a range growing all the way to 64 executors. The other 41% of the Spark applications that do not have Dynamic Allocation enabled by default have the default executor count as shown in Figure 3b. We can see that 80% of these applications that do not have Dynamic Allocation enabled run with a default executor count of 2. Overall, we observe that the default resource settings for Spark SQL queries are far from



(a) Queries per application **(b) Variation in application queries** **(c) Concurrent applications within clusters**

Figure 2: Insights from production Spark workloads at Microsoft.



(a) Non-default range for Dynamic Allocation **(b) Static resource allocation** **(c) Optimal executors for TPC-DS queries**

Figure 3: Executor counts in production Spark workloads and optimal executor counts for TPC-DS.

ideal, with unrealistic minimum and maximum values set for the executor counts.

2.3 Reactive Approaches

Spark's Dynamic Allocation [10] reacts to tasks piling up during the course of query execution and allocates exponentially more executors to improve the performance. While it can indeed help with unexpected number of tasks, it takes some time for the additional executor requests to be fulfilled, and might require several requests before the required number of executors are finally allocated. Therefore, it runs the risks of allocating too late as well as exponentially overshooting the required count. Users can control the minimum and the maximum executor counts with Dynamic Allocation, however as we discussed above, the default values are set to 0 and $2^{31} - 1$ respectively.

Sparklens [5] allows users to analyze executor count usage of a Spark SQL query in a post-hoc manner, i.e., reacting after a query has finished executing, and get recommendations for better executor count allocations for future instances of the same query. Still, the approach is limited to improving the resource allocation for the same query and inputs, and requires analysis for every such query, which may not be possible for large production workloads. Other approaches such as Tuneful [35], Reloca [39], and Perforator [50] rely on sample runs to generate training data and build models for predicting the resource requirements.

Cloud analytics platforms today provide auto-scaling of the compute nodes, i.e., reactively expand or shrink the total number of nodes available to a customer. Theoretically, if such auto-scaling is fast enough then it can react to per-query requirements by allocating or de-allocating the overall resources available, i.e., if more tasks get queued up and Dynamic Allocation cannot find anymore available executors, then auto-scale can kick in and allocate more nodes. In practice, however, allocating VMs can take several minutes [26], thus taking a while before the auto-scale can react to resource needs at the query-level.

Finally, many analytics platforms also have auto-pause capability to stop charging customers if there are compute resources have been idle for some time [2, 8, 16, 49]. However, similar to auto-scale, auto-pause could take several minutes to react and comes into effect only when there are no queries by the user or tenant. Furthermore, auto-pause is even more conservative since the paused nodes will take a few minutes to be resumed making it ineffective for short pause intervals.

2.4 Prediction Challenges

Predictive resource allocation at the query level is challenging. Figure 3c shows the distribution of the optimal number of executors over different TPC-DS queries for four different scale factors. We can see that the optimal values vary for different queries and also for the different scale factors, varying from as little as 1 executor all the way to 48 executors, and thus indicating that a rich set of features containing both the query and data characteristics are needed to predict the resources at the query level. Larger scale factors (larger data sizes) tend to have more queries with larger values of the optimal executor counts. For example, the optimal count is ≥ 16 for 56 queries at SF=1 and 10 in contrast with 75 and 88 queries for SF=100 and 1000 respectively. The optimal count is ≥ 32 for 16, 25, 32, and 45 queries for SF=1, 10, 100, and 1000 respectively.

3 PRICE-PERFORMANCE MODEL (PPM)

We now present our approach to price-performance model (PPM). We build on top of prior work on predicting optimal resources for SCOPE jobs at Microsoft [47, 48], and extend it to a more generalized framework for price-performance optimization.

3.1 Model Framework

Our approach to selecting the optimal configuration involves first predicting the PPM, that is, the relationship between resource allocation and execution time and then selecting the optimal

configuration according to the price-performance optimization objective. We can use the same predicted PPM to select different configurations that optimize for various objectives without needing to re-predict the PPM separately for each scenario. There are two components to our modeling approach that are defined as follows.

(1) Represent the PPM by a mathematical function with known properties. It is parametrized by scalars whose values depend on the query characteristics. The time $t(n)$ taken by a query with n executors is given by:

$$t(n) = f(n, \{scalar\ parameters\}) \quad (1)$$

(2) Train a parameter model to learn the values of the scalar parameters of f for a given query:

$$g : \text{query characteristics} \mapsto \{scalar\ parameters\} \quad (2)$$

This model is used to predict the parameter values of f for a newly-submitted query. Note that the parameter model is scored only once per query, not once per candidate configuration, and $t(n)$ is estimated by evaluating the predicted instantiation of f at different values of n. We discuss the parameter model in more detail in Section 3.4.

Similar to prior work [47, 48], we impose a condition of monotonicity while selecting candidate functions for f. This condition means that $t(n)$ should be monotonically non-increasing with n. This is consistent with user expectations that a query's run time should not increase with more resources allocated to the query.

In practice, this expectation can be violated in systems, e.g., due to parallelism overheads on small sizes or skew in input data. However, we still impose the monotonicity constraint on the PPM model due to the following reasons – (1) It is never cost-efficient to operate in a region where time increases with allocated resources, so accurately modeling that behavior is not needed; (2) even in cases when non-monotonic behavior is observed, the overall minimum time often is the same or close to the minimum time in the initial monotonically decreasing region, so optimization objectives relative to minimum times would not be affected; (3) run-time estimates from Sparklens, that we use to extract parameters for training our models (Section 3.4), are always monotonically non-increasing; (4) being consistent with user expectations with simple monotonic models helps with explainability of our resource model decisions and with forecasting the future resources provisioning needs.

We evaluate two candidates for the PPM function f as follows.

Power Law with saturation: We leverage the performance characteristic curve from the prior work [47, 48], which uses a power-law function for f, but also extend it by adding a constant term m that reflects a lower bound on the running time of the query. The PPM model is thus formulated as:

$$t(n) = max(b \times n^a, m) \quad (3)$$

This model has three query-specific parameters, a, b, and m, that the ML model will learn and predict. We abbreviate this model by AE_PL in the rest of this work.

Amdahl's Law: This is inspired by the well-known Amdahl's Law model for computation speedup with increase in allocated resources [14]. In this model, the latency is divided into two parts: a fixed component s that is invariant to changes in resource allocation and a scalable component that is inversely proportional to the amount of resources. The PPM model is thus formulated as:

$$t(n) = s + \frac{p}{n} \quad (4)$$

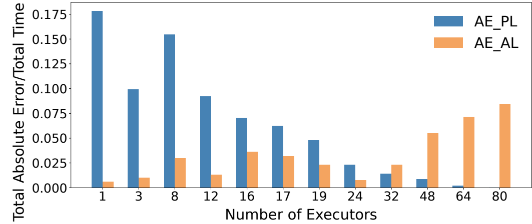


Figure 4: PPM model errors for different executor counts for two models, AE_PL and AE_AL, when fit on Sparklens estimates of the PPM over all queries of TPC-DS SF=100.

This model has two query-specific parameters, s and p, that the ML model will learn and predict. We abbreviate this model by AE_AL in the rest of this work.

For AE_AL, saturation happens at (infinitely) large values of n for parallelized queries rather than at smaller query-specific values unlike the AE_PL model. While saturation constraints could be refined for AE_AL, we use the Amdahl's Law model as is, since it is a well-established and very popular model. But its lack of job-specific early saturation limits its applicability to this domain, in particular, it can cause resource allocations to far overshoot optimal values when the optimization objective is to minimize executor counts subject to a maximum allowed slowdown [Section 5.3].

3.2 Model Comparison with Sparklens

We now compare the accuracy of the above two predictive PPMs with Sparklens, which is a post-hoc reactive model after the queries have finished executing. While Sparklens simulates the Spark scheduler, it implicitly follows a model of the query run time by first determining the length of the critical path and then distributing the remaining tasks according to available executors. Conceptually, this is similar to the Amdahl's Law approach except that the scaling of time with resources need not be uniform and saturation happens, that is, time estimates stop decreasing beyond some point.

Figure 4 shows how well the AE_PL and AE_AL models fit Sparklens estimates for all TPC-DS queries at SF=100 for different values of n. The Sparklens estimates were obtained from a single run of each query at $n = 16$. We see that AE_AL is a better fit to Sparklens estimates for $n < 32$ while AE_PL is a better fit beyond that. Thus, one can obtain an error of around 5% or less for the full range of n by using the models over different ranges of n. Interestingly, we see that although AE_AL fits Sparklens estimates better at lower values of n, it does not translate into better prediction accuracy than AE_PL when compared to actual query run times in a number of cases, as we will show in Section 5.

3.3 Impact of Total Cores

So far the PPM (Equations 1– 4) has considered the number of executors, n, as the only input parameter representing the computational resources available to the query. However, the total number of compute cores, k, is also determined by the number of cores per executor, e_c, since $k = n \times e_c$. To allow for different values of e_c, one approach could be to modify the PPM to consider it as an additional parameter, but this increases the model complexity. Instead, we can directly use k in the PPM as we discuss below. The model's ability to work with total cores makes it more applicable since it can now be used on datasets with different e_c values for queries.

To explore this idea, we evaluated several configurations with different values of n and e_c (six configurations with $e_c \neq 4$), for each TPC-DS (SF=100) query. We estimated the time for $e_c \neq 4$ configs using the (linearly-interpolated) time with $e_c = 4$ for the same value of k, and compared it with the actual time for $e_c \neq 4$. The relative estimation error, calculated as $1 - \left(\frac{t_{n,e_c \neq 4}}{t_{n,e_c=4}}\right)$, were small, with an average of 8.8%, and with 68.4% and 92.9% of the points lying in the intervals [-10%,10%] and [-20%,20%] respectively. Overall, using k for the PPM instead of n and e_c separately gives good accuracy while reducing model complexity.

There may be several choices for factorizing the optimal k for a query into n and e_c values. Choosing smaller values of e_c offers more granularity in cost-performance trade-offs due to the larger range of possible values for n. Since currently executors cannot span multiple nodes, we want to choose executor sizes that minimizes resource wastage (stranded resources on a node) to minimize the total nodes, r. For example, assume that each node is homogeneous, has C cores, M amount of memory, and each executor will get e_m memory. One may solve an optimization problem to minimize r such that

$$e_m \times \lfloor \frac{C}{e_c} \rfloor \leq M, \; r \times \lfloor \frac{C}{e_c} \rfloor = n, \; and \; e_c \times n = k.$$

Additional considerations can constrain the factorizing strategy, such as reducing garbage collection overheads with very large e_c and avoiding difficulties in determining the optimal amount of overhead memory with very small e_c values [4].

3.4 Training Parameter Model

The goal of the parameter model is to learn a function g : query characteristics $\mapsto \{a, b, m\}$ for the AE_PL model or g' : query characteristics $\mapsto \{s, p\}$ for the AE_AL model, depending on the choice of the PPM. Section 4 discusses how we extract the query characteristics. For model training, we additionally need the PPM parameters as targets (labels). On the other hand, model scoring will predict the PPM parameter values, and thereby also the PPM. We use an off-the-shelf implementation [23, 46] of Random Forest regression models for the parameter model.

For training the parameter model, the PPM parameters are obtained by fitting the PPM to run times of each query for different configurations, which in our case is the number of executors, n, or number of cores, k. The run times may be from actual query runs or estimates of run times from simulators or other tools. Getting actual run times for different configurations, along with multiple runs to account for run-to-run variance, can be time consuming (also see Section 5.1). Moreover, rerunning queries with different configurations may not be easy for production workloads. Instead, our approach is to run the training queries once (at $n = 16$) and use estimates generated by Sparklens with a post-execution analysis on the logs. Sparklens can also generate estimates from production workloads. Note that we are using simulation to augment the training data for speed and convenience in production environments, but our models can also be trained with actual run time data for all configurations in case they are available.

Once the training data is available, the PPM parameters for each query can be obtained as follows. For AE_PL, the power-law portion of the PPM can be transformed into logarithmic space as:

$$log(t(n)) = log(b) + n \times log(a) \quad (5)$$

Table 1: Feature list for parameter model

Feature	Description
# Aggregate, Project, Join, Filter, Sort, Union, etc.	Count of each type of operator in the query plan (14 operators for TPC-DS)
$\sum$ all operators	Total number of operators in the query plan
Max Depth	Maximum depth of query plan
# Input sources	Number of input data sources used by the query
$\sum$ Input bytes	Estimated total number of bytes of input data used by the query
$\sum$ Rows processed	Estimated total number of rows processed by all operators in the query

$log(a)$ and $log(b)$ can then be determined by fitting a linear regression model to $log(t(n))$ as a function of n. For this, we consider only the non-saturating region for $t(n)$, that is, over the region $n \in [1, n_m]$ where $t(n_m) = m$ for all $n > n_m$. We determine m by the minimum run time of the query seen over all configurations. For AE_AL, we determine s and p by fitting a linear regression model to $t(n)$ as a function of $\frac{1}{n}$. The parametric approach thus compresses data points into an $O(1)$-space representation per query.

Table 1 shows the features used for the parameter model. This includes characteristics for the query plan as well as inputs to the query, and is motivated by our observation that the optimal executor count depends on both of these aspects (also see Figure 3c). We only use features that are available at compile-time and optimization-time of the query since (1) we want to predict the optimal executor count *before* running the query and (2) we need to use the *same* features for scoring the model as we used for training it. Thus, we do not include any runtime statistics as features for the parameter model. We evaluate model feature importances in Section 5.7.

With our parametric PPM approach, we construct a single training data point for each query in our training dataset, regardless of how many different configurations for which the query run time is available. Thus, if we are training over all 103 queries of TPC-DS, our training dataset will have 103 data points. During model scoring time, the model is scored only once per query regardless of the number of candidate target configurations; the predicted times for the target configurations are determined by evaluating the PPM functions (Equations 3 and 4) which are generally much faster than model scoring times except those for simple, linear models. Section 5.6 discusses the overheads.

In contrast, a non-parametric approach would include run times for every configuration of each query as a separate data point in the training dataset. So, for the above example, the training dataset would have $103 \times c_{tr}$ data points where c_{tr} is the number of training configurations for each query. If there are c_{tt} candidate configurations for each test query, then it would score the model c_{tt} times as opposed to only once with the parametric approach. Thus, our parametric PPM approach reduces training datasets, and subsequently random forest model sizes, model training and scoring times compared to a non-parametric approach.

4 *AUTOEXECUTOR* INTEGRATION

We now describe how we integrate predictive price-performance optimization with Spark query engine [15]. Traditionally, the

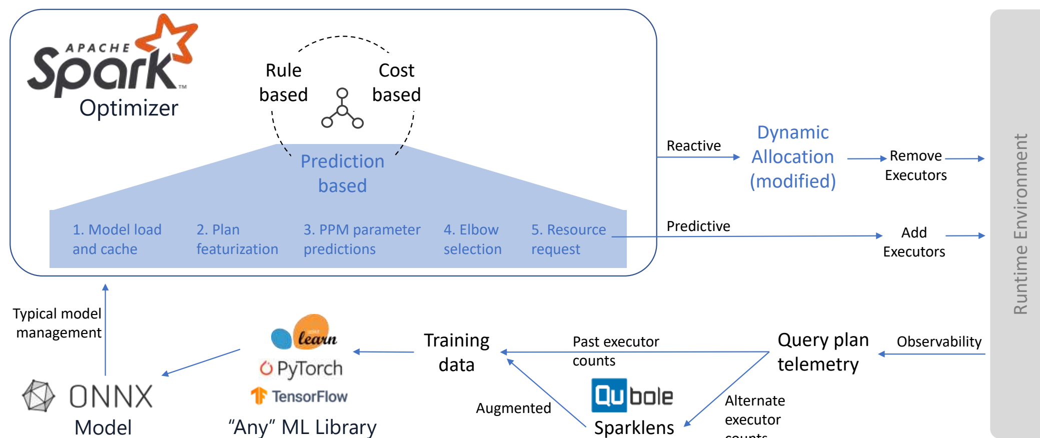


Figure 5: The *AutoExecutor* system design illustrating the prediction-based resource optimization in Spark.

Spark query optimizer performs rule-based and cost-based optimizations, and provides extensions for custom rules and cost models. We augment the Spark optimizer to support *predictive optimizations*, i.e., using ML-models to make the optimization decisions. These models could be trained offline using any of popular ML libraries such as Scikit-learn, PyTorch, TensorFlow, etc., and we score them efficiently and in-process during the course of query optimization. We believe this a major shift for query optimization in Spark and while we focus on predictive executor counts in this paper, our approach can be leveraged for many other predictive optimizations in the future. Figure 5 shows our augmented Spark optimizer with support for predictive query optimizations. Below we walk through the different components in this architecture.

4.1 Training Data

We collect a rich set of data from past query runs to generate training data for *AutoExecutor*. The training features for *AutoExecutor* include query characteristics, input dataset information, and runtime statistics as shown in Table 1. To collect this information, we use Peregrine [40] and SparkCruise [51] to log detailed plans with annotations such as input dataset information, and runtime metrics at the end of every query. The collected data is transformed into a tabular representation of the query workload. The table contains one row per query.

Given that past telemetry contains runtime metrics for a given executor count, with which the query actually ran, we need more data with different executor counts in order to train the PPM. We achieve this by augmenting the past executions with simulated runs for other executor counts using Qubole Sparklens [5]. Sparklens simulates the Spark scheduler to provide expected runtimes with different executor counts. These simulation points provide additional training data for the same query. Another alternative is to re-run the queries with different executor counts. This method is more expensive and might not be possible for production workloads.

4.2 Model Training

Once we have the training data that has been augmented for different executor counts, we train the parameter model described in Section 3.4. *AutoExecutor* allows using any of the popular ML libraries, like Scikit-learn, PyTorch, TensorFlow, etc., as illustrated in Figure 5. Although, we used Scikit-learn in this paper, this flexibility is useful in improving the models over time for different workload characteristics by trying out different libraries. We describe the training environment and overheads in Section 5.6.

4.3 Model Format

The Spark optimizer code runs inside Java Virtual Machine (JVM). However, data scientists heavily rely on Python libraries, esp. Scikit-learn. Currently, the *AutoExecutor* pipeline also uses Scikit-learn library to train models. This language barrier makes it difficult to use making it difficult to use Scikit-learn models for inference inside the optimizer. To solve this interoperability problem, we convert the models into ONNX format [21]. Scikit-learn, like many machine learning libraries, supports converting models into ONNX format. ONNX model runtime provides Java bindings and can be used inside Spark optimizer. We can also replace the training library with TensorFlow, PyTorch, etc. as long as they also export to the ONNX model format. Once the ONNX model is produced, we can leverage typical model management libraries and infrastructure, such as Azure Machine Learning [7] or MLflow [56]. Additionally, ONNX model runtime has multiple optimizations to improve the inference time. *AutoExecutor* requires fast inference times as it runs inside the query optimizer and any delay will affect the end-to-end query completion time.

4.4 Model Scoring

AutoExecutor introduces prediction-based optimizations in the Spark query optimizer and applies a series of five steps, as illustrated in Figure 5, to request the desired number of resources during optimization phase before a query is run. We implemented the prediction-based optimizations using the Spark extensions feature [1] and it does the following. First, we load the ONNX model from the corresponding model register, e.g., the AML model registry. In contrast to traditional model scoring, we load the model into the optimizer process for low-latency scoring. We also cache the models once loaded inside the optimizer to not load them repeatedly since the inference step is in the live query path. We evaluate the performance of inference inside optimizer in Section 5.6. Then, we featurize the optimized query plan and its input datasets into a feature vector and feed them to the parameter model to get the predicted parameters for the PPM. With the predicted PPM parameters, *AutoExecutor* rule gets the execution time predictions for different number of executor counts. The default executor selection strategy automatically selects the

executor count right before the performance flattens in the price-performance trade-off to achieve the fastest query performance with minimal number of executors. This strategy can be updated depending on the price-performance tradeoffs of the user.

4.5 Predictive Resource Allocation

After using the model to get the desired executor count, *AutoExecutor* automatically requests for resources before the query is run and releases the resources after the query has finished. We use the executor allocation API in Spark to request additional number of executors from the cluster manager. Note that the allocation request is not binding and the cluster manager might allocate fewer than the requested count depending on available resources.

4.6 Combining Predictive and Reactive Approaches

Finally, we need to combine the predictive resource allocation with the reactive approach, namely the dynamic allocation. Specifically, we use dynamic allocation as a mechanism to release excess resources after they are no longer needed by the query, i.e., to de-allocate resources. The reasoning behind this is long-running analytics queries typically require more resources upfront for more scale-out to process larger volumes of data before they get filtered or aggregated [29]. *AutoExecutor* uses a modified version of dynamic allocation strategy from Spark. We disable dynamic allocation for scaling up since we can predict the resources upfront, but enable dynamic allocation to remove executors after they have been idle for more than a specified time duration.

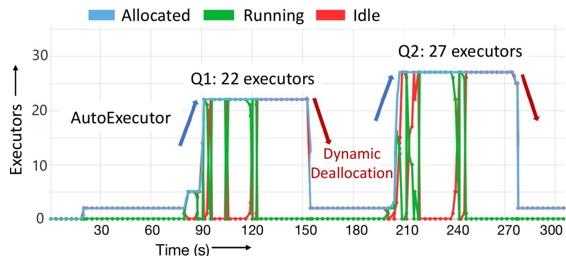


Figure 6: Combining predictive and reactive allocation for individual queries in an interactive Spark application.

Figure 6 shows *AutoExecutor* with predictive allocation and reactive deallocation of executors for two queries in an interactive Spark notebook. When the first query is submitted, *AutoExecutor* predicts $n = 22$ and automatically requests it. We see the completed executor allocation at around 90 seconds in Figure 6. There is a time gap between the end time of first query and the submit time of second query. During this time gap, dynamic (de)allocation releases the idle executors. For the second query, *AutoExecutor* automatically allocates the predicted executor count of 27. We release the resources after both the queries have finished.

5 EVALUATION

We now present an experimental evaluation of *AutoExecutor*. Our goal is to answer several key questions, including how good are the models, what is the impact of price-performance trade-off, how much cost savings can our predictive approach bring, can predictive approach cope with changes in input sizes, what are the overheads involved, and which features are more important than others. We first describe our setup, then discuss each of the above questions.

5.1 Setup

We use TPC-DS [27] for evaluation since it is the most popular benchmark for analytical workloads, both in academia and in industry [31, 43]. It allows us to easily scale input data sizes by orders of magnitude, compare performance with current state-of-art systems such as Sparklens [5], and provides a repeatable baseline for future work in this area.

Our testbed consists of 103 TPC-DS queries (99 queries + variants) [25], for four scale factors SF=1, 10, 100, and 1000 running on Azure Synapse Spark pools with medium-sized nodes (8 cores and 64 GB memory per node). At most two executors can be placed on each node. We allocate 4 cores ($e_c = 4$) and 28 GB memory for the driver and each executor. We vary the number of executors, n, from 1 to 48. All other configuration parameters are fixed for these experiments. Each TPC-DS query runs as a separate application and we record its time elapsed, that we refer to as $t(n)$ in this work.

We run each query several times for every $n = 1, 3, 8, 16, 32, 48$, then take averages after discarding outliers (points lying outside ±1.5× the inter-quartile range. The 'Actual' series shown in the figures correspond to this averaged run time data. Gathering ground truth data using actual runs is thus time-consuming and expensive, not only due to the need to run queries with different configurations, but also to do repeated runs to get reasonable averages. Fast and deterministic simulation tools, such as Sparklens, are thus quite useful for data augmentation to train ML models. For each query, we obtain Sparklens estimates for the application time, shown as series 'S' in the figures, by running the tool on the executor logs for a single run of that query with $n = 16$. Depending on the SF, total $t(n = 16)$ over all queries was 8.5%–13.5% of total $t(n)$ over all the n listed above, showing the savings potential in training data collection costs by using simulation. Of course, the testing dataset only uses runtimes from actual runs.

We used TPC-DS since it is the most popular benchmark for analytical workloads both in academia and in industry. We evaluated model generalizability along two dimensions: (1) data sizes by changing scale factor (SF), with different train and test SF, and (2) query templates—we do a 5-fold cross validation (80:20 training:test dataset split) and repeat it 10 times. For each repeated trial, the 5 folds collectively cover all of the TPC-DS queries in the testing datasets while not including any test query in the training dataset. We evaluate two models for AutoExecutor, Power Law and Amdahl's Law, that are referred as AE_PL and AE_AL series respectively in the figures.

5.2 Time Prediction

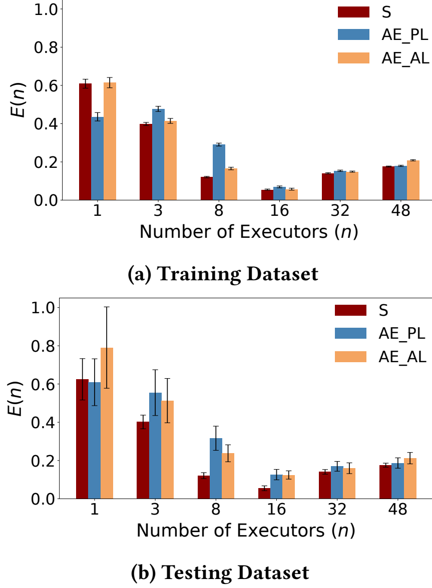
We now evaluate how well the run times can be predicted for different values of n. For the experiments we use SF=100 and evaluate model generalizability for query templates by splitting the dataset and doing cross validations as described above. We also compare against estimates from Sparklens. However, to get the Sparklens estimates for the testing dataset, we need to execute each test query once, which we do with $n = 16$. We fix $e_c = 4$ for all configurations.

To determine the overall prediction accuracy for the test dataset, we compute the following error metric. Let $t_q(n)$ and $\hat{t}_q(n)$ denote

the actual and predicted run times for query q with n executors.

$$E(n) = \frac{\sum_q |\hat{t}_q(n) - t_q(n)|}{\sum_q t_q(n)} \tag{6}$$

$E(n)$ is the ratio of the sum of the absolute time errors to the sum of the actual run times, with the sums taken over all queries in the test dataset. An ideal predictor would have $E(n) = 0$ for all n.



(a) Training Dataset

(b) Testing Dataset

Figure 7: Average errors, $E(n)$, for Sparklens (S), AE_PL, and AE_AL aggregated over test queries from TPC-DS SF=100 (10-repeated, 5-fold cross validations). The error bars show ±1 standard deviation across the $5 \times 10 = 50$ testing folds.

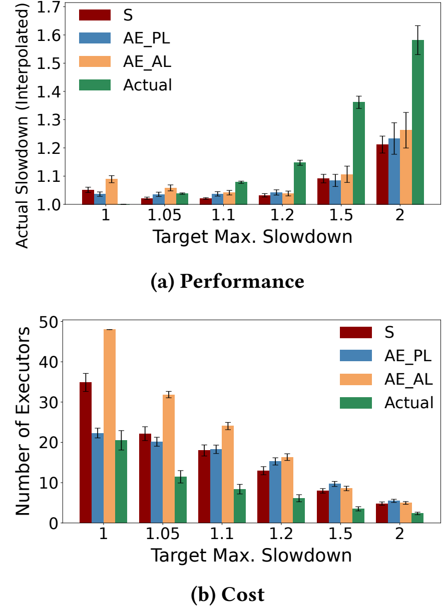
Figures 7a and 7b show average $E(n)$ for the training and testing datasets, for SF=100, corresponding to each fold of the 10-repeated 5-fold cross validations. For both models, as well as for Sparklens estimates, the errors are largest for small n, smallest for intermediate n, and intermediate for large n. This pattern is similar for both training (fit) and testing dataset (prediction) errors. This pattern is affected by our choice of reference configuration ($n = 16$) for invoking Sparklens—a smaller or higher value of n would reduce the errors in that region while likely increasing it elsewhere. The relatively larger errors at small n are not problematic since such n are rarely optimal operating points for optimal performance or balanced tradeoff between increased cost and performance loss ('elbow points', also see Section 5.3).

The model fit and predict errors are close to Sparklens estimation errors. This is because we augment the training data for the PPM model with the Sparklens estimates. The average absolute difference in $E(n)$ values from Sparklens was quite small: 0.079 for AE_PL and 0.087 for AE_AL on this testing dataset.

5.3 Configuration Selection

We now evaluate how well we can select optimal configurations based on run time predictions. We consider two scenarios: (1) cost-savings with a limited slowdown, and (2) elbow point selection. We use SF=100 for experiments in this section. We also piecewise-linearly interpolate the Actual and Sparklens series to cover all $n \in [1, 48]$ and thus expand the set of target configurations.

Limited Slowdown: The goal in this scenario is to select the *smallest* n such that the slowdown compared to the minimum



(a) Performance

(b) Cost

Figure 8: Configuration selection impacts for Sparklens (S), AE_PL, and AE_AL aggregated over test queries from TPC-DS SF=100 (10-repeated, 5-fold cross validations). Slowdowns H are with respect to actual run times (piecewise-linearly interpolated). The error bars show ±1 standard deviation across the $5 \times 10 = 50$ testing folds.

time t_{min} does not exceed a threshold H, i.e., $\frac{t(n)}{t_{min}} \leq H$. Figure 8a shows the slowdowns using the (interpolated) actual run times for the selected configurations for different values of H, while Figure 8b shows the corresponding n. We average over the queries in the test datasets for the 10-repeated 5-fold cross validations. The error bars correspond to ±1 standard deviation of averages for the 10 repeats.

$H = 1$ corresponds to the scenario where the smallest number of executors is used to achieve the best performance (no slowdown over t_{min}). We find that with the selected configurations, there is a resulting additional slowdown of 5.1% for Sparklens, 3.6% for AE_PL, and 8.9% for AE_AL. The average values for n are 20.5 for Actual and 34.9, 22.3, and 48 respectively for Sparklens, AE_PL, and AE_AL, thus reflecting a significant savings opportunity. AE_PL improves over both Sparklens and AE_AL and realizes a substantial portion of the savings opportunity while incurring a small additional slowdown (which is comparable to the average variation in run times). AE_AL always select the maximum value of n (= 48) *due to the lack of early saturation* of the model unlike AE_PL.

As discussed in Section 2.2, users often run jobs with static (same settings for all jobs), small values of n. But using the AE_PL or AE_AL models, we can automatically choose a configuration that optimizes for $H = 1$ as above, to provide significant speedups. The above selected configurations provided an average speedup of 39.5–43.8% for static $n = 3$ (12 cores) and 3.4–7.9% for $n = 8$ (32 cores). Speedups over static $n = 2$ (8 cores) would be higher (expected 2.1× using interpolated values for Actuals).

For larger values of H, particularly $H \geq 1.1$, the configurations selected by the models, as also from Sparklens, tend to be conservative in exploiting slowdown thresholds and saving executor counts. The average slowdowns for AE_PL for $H = 1.05$, 1.1, 1.2, 1.5, 2 were 1.04, 1.04, 1.04, 1.08, 1.23 for AE_PL (with

average n = 20.2, 18.3, 15.3, 9.7, 5.5), but 1.04, 1.08, 1.15, 1.36, 1.58 (with average n = 11.5, 8.4, 6.1, 3.5, 2.4) for Actual. Note that the slowdown for Actual is less than the target slowdown due to n being a discrete variable. This shortfall varies according to queries, and hence to composition of the test dataset in each fold of the 5-fold cross validation. Overall, AE_AL tends to significantly overestimate n. While AE_PL realizes only a part of the full savings potential, it has a similar impact as that of Sparklens but is able to achieve it without executing the query as opposed to post-execution analysis by Sparklens.

'Elbow Point' Selection: We see in the example curve of Figure 1 that the PPM has two distinct regions: at low executor counts n, the run time $t(n)$ changes rapidly for a small change in n, whereas at high values of n, $t(n)$ changes slowly reflecting a diminishing return on investment for increasing n. In this configuration selection experiment, we aim to select the 'elbow point' that strikes a balance between rate of decrease in $t(n)$ and rate of increase in n.

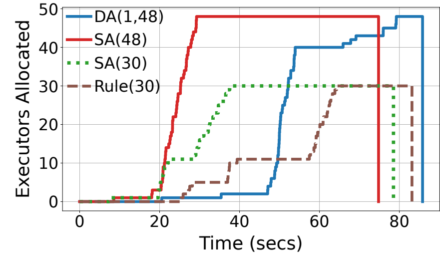
Note that in these example curves, the x- and y-axes, corresponding to n and $t(n)$ respectively, are on different scales. However, we need a way to compare the two quantities in order to determine the elbow point. So, we normalize n and $t(n)$ using range-scaling functions $u : n \mapsto [0, 1]$ and $v : t(n) \mapsto [0, 1]$, then compute the slope at each point $u(n)$ of the normalized PPM. The elbow point L is determined as the *smallest* n for which there is a crossover point for the slope, compared to unit slope, in the normalized PPM. That is, the smallest n for which $slope(u(n)) \geq 1$ and $slope(u(n+1)) \leq 1$. Note that the above is only an example; other definitions for the elbow point can be chosen according to user needs and calculated using predictions from the models.

For the above definition, the vast majority of queries have $L = 8$. For SF=100, only 13 of 103 queries have $L < 8$ for Actual while for Sparklens estimates, all but one query had $L = 8$. For model predictions, we consider the test datasets over the 5-fold cross-validations, then take averages over the 10 repeats. AE_PL selected 8–11 for L. Interestingly, AE_AL selected $L = 7$ for all these queries.

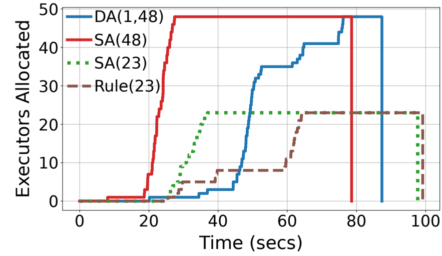
5.4 Cost Savings

Figure 9 shows the executor allocation skylines for two example queries with SF=1000 when run using four policies: dynamic allocation (DA) with n restricted to $[1, 48]$; static allocation (SA(n)), that is all executors requested upfront during job submission, once with $n = 48$ and once with $n = x$; Rule(n), where a request of $n = x$ total executors was made during the *AutoExecutor* optimizer rule for a run that started with $n = 5$. The x executor count was predicted by AE_PL for these queries in one of the 5-fold cross validation experiments with optimization objective of $H = 1.05$ (see Section 5.3). Our evaluation, which gives an advantage to DA, is with hot standbys for nodes. Thus, executors are allocated as fast as possible upon request (also for SA). But for cost (*AUC*) calculations, we only consider executors actually allocated to the query.

Due to the choice of our optimization objective ($H = 1.05$), the queries are expected to take longer to run for SA and Rule with the chosen $n = x$ compared to SA(48), as we see in these examples. The first example (Figure 9a) shows a scenario where SA(x) and Rule(x) took less time than DA(1,48) while the second example (Figure 9b) shows the reverse scenario. Both DA and Rule take more time than the SA policies, with the delay for Rule due to a lag from the time when the *AutoExecutor* optimizer rule made the



(a) Skylines for an example query. *AUC*s for DA(1,48), SA(48), SA(30), and Rule(30) are 1542, 2441, 1516, and 979 respectively.



(b) Skylines for an example query. *AUC*s for DA(1,48), SA(48), SA(23), and Rule(23) are 1641, 2675, 1527, and 1093 respectively.

Figure 9: Executor allocation skylines for two example queries, SF=1000, with Dynamic Allocation (DA), $1 \leq n \leq 48$, Static Allocation (SA(n)) with $n = 48$, and also with $n = 23$ and 30 which are the executor counts requested during optimizer Rule(n) execution for the two queries respectively using AE_PL model and $H = 1.05$ config. selection policy.

request and the full allocation of the requested executors by the runtime environment. However, DA and Rule are not equivalent since Rule requests all executors at optimization time whereas DA reacts gradually over the lifetime of the query execution, after optimization. Current system overheads in fulfilling executor requests by Rule makes its performance similar to that of DA. A more efficient scheduler implementation in future may reduce these overheads. But the delays for DA are more fundamental, due to the need to observe part of the query execution before making its requests. SA policies, while faster, may only take into account query characteristics from prior runs as they need to specify n even before the query is compiled.

Regarding cost savings, we first note that DA, like SA(48), allocates up to 48 executors for these examples. In contrast, our models predicted 30 and 23 executors for these two queries, which is a substantial reduction in the resource requirements with the potential to reduce resource provisioning needs, wait time to reserve resources, and improve cluster utilization by freeing up resources for other queries. We also note a substantial reduction in *AUC* with all policies compared to SA(48). While SA(n) for the predicted $n = x$ slightly reduced *AUC* compared to DA, Rule significantly reduced *AUC* compared to DA: 1542⟶979 and 1641⟶1093 for the two example queries respectively. To restrict DA from over-shooting, one would need to manually provide a query-specific upper bound or use learned models such as *AutoExecutor* to predict and set an upper bound depending on query characteristics.

We evaluated cost savings over all TPC-DS SF=1000 queries, for $H = 1.05$ and with one set of 5-fold cross validation experiments. Rule substantially saved n; the average per-query n was

26.3 for Rule whereas it was 48 for SA(48), that is, 45.1% reduction (median: 50%) in resource allocation requirements with Rule, and 40.8 for DA(1,48), that is, 35.5% reductions (median: 37.5%) with Rule. Over all queries, Rule saved median and total *AUC* by 20.3% and 16.1% over DA, and median and total *AUC* by 65.2% and 54.8% over SA(48).

But the resource and cost savings also comes with a performance loss, beyond expected levels (5% target for these experiments), due to the lag in rule invocation and gradual allocation of executors. The runtime environment takes ~20–30 secs to gradually allocate the requested executor count, and is beyond the control of the prediction models. The median per-query slowdown with Rule was 17.6% compared to SA(48), and 10.3% compared to DA(1,48). Total query time was 16.8% and 10% longer compared with SA(48) and DA respectively. Also, for the above experiments, 12 of the 103 queries finished before the runtime environment was able to completely allocate all executors requested by Rule (AE_PL predictions). The average shortfall for these queries was 5.8 executors whereas their average request was for 22.3 executors. The total allocated executors matched the model-predicted value for the remaining 91 of the 103 queries. For smaller scale factors, a higher percentage of queries are likely to see shortfalls since queries finish faster before the allocations can be fully satisfied by the system.

One potential direction for future work is exploring if the performance losses can be reduced by accounting for the allocation delays in the model predictions and subsequent configuration selections. This would induce selection of a higher executor count to make up for the delays, but would also potentially reduce *AUC* savings. The allocation delays could be estimated by a periodic calibration step.

5.5 Change in input data size

We now evaluate how well the models perform when the test queries operate on a different amount of data than what they had in the training datasets for the models. A simple way to test this is to train the models on one or more scale factors of TPC-DS and then test them on a different scale factor.

Figure 10 shows average prediction errors, $E(n)$, over all 103 queries when the testing dataset is SF=1 and the training dataset is SF=1, 10, 100, 1000 for subfigures (a)–(d) respectively. Thus subfigures (b)–(d) represent a *scale-down* scenario for data sizes from training to test. We see that both AE_PL and AE_AL do better than Sparklens for most cases. The gap increases with difference between the training vs testing SF, with it being most pronounced in Figure 10d where SF=1000 is used for training.

Figure 10 shows the scenario when the testing dataset is SF=1000 and the training dataset is SF=1, 10, 100, 1000 for subfigures (a)–(d) respectively. Thus subfigures (a)–(c) represent a *scale-up* scenario for data sizes from training to test. Here we see that Sparklens does better in most cases than the model predictions but unlike in the previous scenario, the gap is smaller and does not necessarily follow the difference between training and test SFs. Interestingly, when the training SF=1 (Figure 11a), which is the largest difference from the test SF of 1000, the models do quite well and even outperform Sparklens for $n \geq 16$. Over all training SFs, $E(n)$ for AE_PL was less than 0.65 for $n \geq 8$.

Figure 12 shows a scenario where all SFs different from the testing SF were used for training the AE_PL and AE_AL models. Subfigure (d) shows a scale-up scenario, (a) shows a scale-down scenario, while (b) and (c) show an in-range scenario. Note that since Sparklens does not include data size as an input, we cannot use an objective estimate for it in any of these scenarios. For example, should one use the min./max./average/some scaled values, etc. of estimates from prior runs? Instead, we show Sparklens estimates from the corresponding testing SF as a reference, assuming they are obtained *after the query has been run*.

Comparing Figure 12d with Figures 11a–c we see that AE_PL does better with training with the multiple SFs together than with any single SF for $n \leq 16$ and is close for larger n. For example, $E(n = 8)$ for AE_PL is 0.39 in Figure 12d whereas it is 0.46, 0.62, and 0.65 in Figures 11a–c respectively. AE_AL is worse than AE_PL for this example, but does better at $n = 8$ in Figures 12a–c. Both models do better with combined training SFs than the worst-case single SF accuracy for all n, for scale-up (Figure 12d vs 11a–c) and scale-down (Figure 12a vs 10b–d). Overall, the model accuracy is reasonable with data size changes for scale-down, in-range, and scale-up scenarios with multiple SFs included in the training dataset.

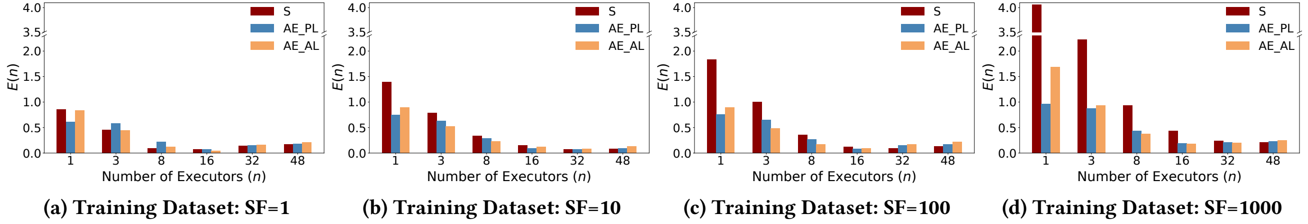
5.6 Overheads

Training: The time taken to fit the parameter model on the Sparklens estimates, as described in Section 3.4, was ~0.3 msec on average for each training data point. As we have discussed, our parametric PPM approach reduces overheads by design. We used scikit-learn's default parameter settings of 100 estimators [23] for the Random Forest model. The pickled file size on disk when trained over all 103 TPC-DS queries (for a given scale factor) was 0.8 MB for AE_AL and 0.9 MB for AE_PL. The ONNX file size was slightly larger at 1 MB and 1.1 MB respectively. The average single-threaded training time for this dataset was ~79 msec.

Scoring: The time taken to predict the PPM using the Random Forest model was on average ~3.6 msec for the scikit-learn model. Inside the query optimizer, the plan featurization time was ~10.3 msec. The one-time cost to load and setup ONNX model for prediction was ~88.1 msec and ~47.1 msec respectively. The ONNX model inference time per query was ~0.9 msec. The *AutoExecutor* optimizer rule is the last rule invoked once per query.

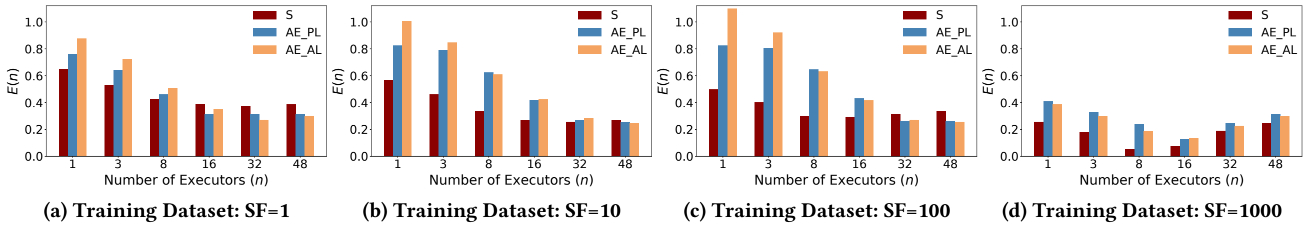
5.7 Feature importance

To understand how different features contribute to model prediction accuracy, we computed feature permutation importance scores [22]. Figure 13 lists the top 8 features in decreasing order of the sum of average importance scores for the model predictions on the testing datasets for SF=100 (5-fold cross-validation, repeated 10 times). The most important features are the estimated total input bytes and rows processed, maximum depth of the query plan, number of operators, and specific operators such as Project and Filter. The results support our view that both amount of input data and query characteristics involving information about the query plan and operators are important for query run time prediction. Some other operators, not shown in Figure 13, appeared much less frequently or changed less in value across queries in our dataset and got low importance scores. We validated this result by retraining the models with this reduced feature set and testing their predictions. We found that $E(n)$ differed from the earlier values (from models trained with the full features set) by less than 0.02 for both AE_PL and AE_AL models for all our values of n.



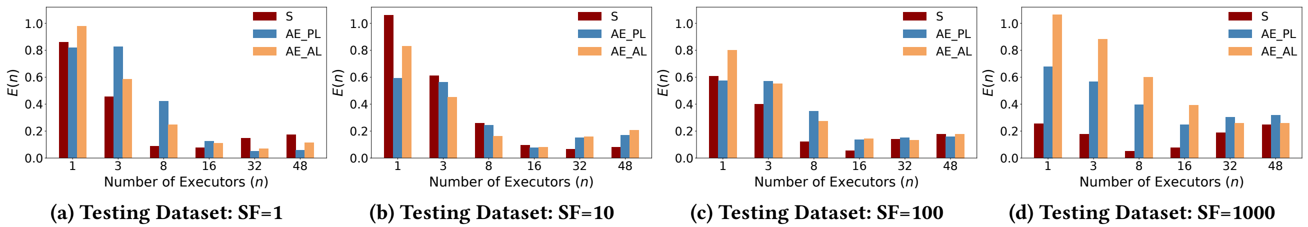
(a) Training Dataset: SF=1 (b) Training Dataset: SF=10 (c) Training Dataset: SF=100 (d) Training Dataset: SF=1000

Figure 10: Testing Dataset: SF=1. Average errors, $E(n)$, for Sparklens (S), AE_PL, and AE_AL, aggregated over test queries.



(a) Training Dataset: SF=1 (b) Training Dataset: SF=10 (c) Training Dataset: SF=100 (d) Training Dataset: SF=1000

Figure 11: Testing Dataset: SF=1000. Average errors, $E(n)$, for Sparklens (S), AE_PL, and AE_AL, aggregated over test queries.



(a) Testing Dataset: SF=1 (b) Testing Dataset: SF=10 (c) Testing Dataset: SF=100 (d) Testing Dataset: SF=1000

Figure 12: Average errors, $E(n)$, for Sparklens (S), AE_PL, and AE_AL, aggregated over test queries. The Training Dataset for AE_PL and AE_AL comprised of data from the 3 other SFs for all queries. Training and Testing Datasets are always separate for AE models. The Training Dataset for Sparklens is the same as the Testing Dataset, and is obtained *after* one run of each query.

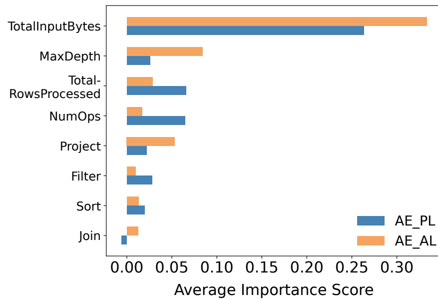


Figure 13: Top 8 features, ranked by AE_PL + AE_AL scores, using permutation importance for the testing datasets from TPC-DS SF=100 (10-repeated, 5-fold cross validations). For each model (fold), we repeat feature permutations 100 times. The averages are over $10 \times 5 \times 100 = 5000$ scores.

6 RELATED WORK

Resource Prediction For Jobs: There has been prior work on predicting resources for big data applications. Tuneful [35] learns over multiple iterations of the query to predict optimal resources and needs tens of iterations to converge. It also does not generalize to new queries. In contrast, our goal is to do a one-shot prediction for both existing and new queries. Reloca [39] relies on sampling runs to generate training data for deep neural network (DNN) models but does not optimize for cost or compares against reactive allocation. In *AutoExecutor* we use simple parametric models that offer better explainability and visualization compared to DNNs. Sparklens [5] does post-execution analysis and cannot handle changes in queries or inputs. *AutoExecutor* can utilize Sparklens estimates to augment data for training ML models. Many systems [37, 38, 42] were designed to optimize the performance of MapReduce applications. For example, Starfish [38] uses dynamic tracing to build profile of MapReduce applications. This application profile is then fed to a what-if engine that can predict its performance over different configurations using analytical and simulation models. In contrast, *AutoExecutor* is integrated closely with Spark and does not require intrusive profiling of running applications that can affect runtime performance. Unlike analytical models that require manual updates to reflect the current implementation of the data processing engine, *AutoExecutor* is resilient to changes in the Spark engine. The machine learning model of *AutoExecutor* can automatically learn changes in performance characteristics from new training samples.

We also briefly distinguish *AutoExecutor* from some of our prior works on resource allocation and predictions. AutoToken [52] predicted the peak parallelism/resources for recurring overallocated SCOPE queries. Unlike *AutoExecutor*, it did not predict performance or optimal resource allocation. Optimal (depending on the metric) allocations can be less than the peak. Fan, et al. [34] used ML models to predict the optimal parallelism/resources for SQL Server queries. They used a non-parametric ML approach, with the model being scored once per target configuration. In contrast, *AutoExecutor* uses a parametric approach, scores the model once per query, and shows improvements over reactive

allocation. Bag, et al. [29] determined when resources allocated to a running SCOPE query are no longer required for the remainder of the run and releases/deallocates those resources early. In contrast, *AutoExecutor* focuses on upfront optimal resource allocation. QROP [55] presented a vision for the query optimizer and resource manager to work together for efficient query processing, but did not propose a predictive mechanism for price-performance optimization.

TASQ [47, 48] predicted the optimal parallelism/resources for SCOPE queries. They used a parametric ML model to predict a performance characteristic curve. The model is scored once per query and they also compare against scoring other models once per configuration. *AutoExecutor* differs from TASQ in taking resource optimization from a SCOPE batch processing environment in Cosmos to an interactive Spark query processing environment in Synapse. Concretely, it differs in the following ways.

- *Model*: *AutoExecutor* extends the prior parametric model by adding a saturation constraint, that enables prediction of where the performance curve flattens with increasing resource allocation, instead of having to rely on user inputs, visual inspection, or ad-hoc strategies. We propose a general parametric framework for price-performance optimization, evaluate the well-known Amdahl's Law model in this context and show its limitations for configuration selection scenarios due to the lack of early saturation, show how to automatically determine elbow points for price-performance trade-offs, and discuss feature importance.
- *Platform*: TASQ addressed optimal allocation for SCOPE queries on Cosmos, which is internal to Microsoft, whereas *AutoExecutor* focuses on Spark SQL queries. The underlying system details are quite different from SCOPE and therefore we had to come up with novel system and algorithm engineering to bridge that gap. In the current paper we described our integration for Spark in depth, including how to leverage an open-source simulator (Sparklens) for data augmentation, how to combine predictive allocation with dynamic deallocation, and how to improve over dynamic allocation. We implemented the end-to-end system to collect Spark telemetry, train simple and interpretable machine learning models with good accuracy, and extended the Spark optimizer to load models in a universal format for fast scoring during optimization and to perform automatic resource allocation. We also show (first, to the best of our knowledge) distributions of resource allocations over many Spark queries on a commercial platform, that highlights the opportunity and importance of optimal resource allocation for Spark.
- *Predictive vs Reactive*: In contrast to all the above prior works, in this paper we compared between the predictive and reactive allocation approaches. Adaptivity has long been thought to fix many of the query processing deficiencies, and techniques such as adaptive indexing, etc. have been presented over the years. Spark introduced Dynamic Allocation on similar lines, and yet in this paper we showed significant cost savings over Dynamic Allocation for Spark.

Workload Prediction: There have been a number of recent efforts to predict load patterns at a cluster or service level. For example, Seagull [49] uses forecasting techniques to predict workload patterns. This approach has many applications such as automatic resizing of clusters and scheduling maintenance tasks but cannot be used to allocate resources at the job level. Amazon announced a predictive scaling policy in EC2 using ML models [12], but the modeling details are not public.

Centralized Job Scheduler: Another relevant area of research is resource allocation techniques employed in job scheduling systems such as YARN [54]. Our approach is complementary to resource allocation techniques that take into account factors like job priorities, wait times, and shared resources. After *AutoExecutor* requests the desired executor count, the scheduler can use any scheduler policy like hierarchical queues [3] or Dominant Resource Fairness [36] to allocate from the pool of available resources. Other comparable techniques to *AutoExecutor* such as Dynamic Allocation [10] also rely on the job scheduler for allocation of resources.

Concurrent Workloads: Prior work has discussed modeling concurrent workloads [44] and sharing resources with performance guarantees [45] for transactional databases. *AutoExecutor* is particularly well suited for analytical systems designed to leverage the cloud's elasticity. In this paper, we have focused on Azure Synapse [6] that reduces contention between shared jobs by utilizing dedicated resource pools and disaggregated storage service [19]. But, *AutoExecutor* is equally applicable on newer cloud databases like Snowflake [24] that also use virtual compute warehouses [33] and storage services to avoid interference between workloads.

7 CONCLUSION

We presented a novel approach for predictive price-performance optimization in analytical queries. Our system, *AutoExecutor*, predicts a parametric model for estimating Spark SQL query run times, and picks a better resource configuration upfront during query optimization. We focus on simple parametric models due to their ease of interpretability, efficiency in compressing training datasets, and enabling fast predictions by requiring only one model scoring per query instead of one per configuration. *AutoExecutor* can optimize for different objectives and is integrated with the Spark optimizer where it considers both the query characteristics and the input data sizes for predicting executor counts. Our extensive evaluation over TPC-DS workloads show that *AutoExecutor* can achieve prediction accuracies very close to Sparklens estimates, which are post-execution, and yet save executor occupancy compared to state-of-the-art Dynamic Allocation in Spark.

ACKNOWLEDGMENT

We thank Rui Fang, Jeff Zheng, Xiaolei Liu, Ruiping Li, Carlo Curino, and other members of the Synapse and GSL teams who provided feedback or useful directions for the *AutoExecutor* project, and the anonymous reviewers for their feedback on this paper.

REFERENCES

[1] 2017. *Add hooks and extension points to Spark.* Retrieved February 12, 2022 from https://issues.apache.org/jira/browse/SPARK-18127
[2] 2018. *New – Predictive Scaling for EC2, Powered by Machine Learning.* Retrieved February 12, 2022 from https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/
[3] 2019. *Hadoop: Capacity Scheduler.* Retrieved February 12, 2022 from https://hadoop.apache.org/docs/r2.9.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html
[4] 2020. *Part 3: Cost Efficient Executor Configuration for Apache Spark.* Retrieved February 12, 2022 from https://medium.com/expedia-group-tech/part-3-efficient-executor-configuration-for-apache-spark-b4602929262
[5] 2020. *Qubole Sparklens tool for performance tuning Apache Spark.* Retrieved February 12, 2022 from https://github.com/qubole/sparklens v0.3.2.
[6] 2021. *Apache Spark in Azure Synapse Analytics.* Retrieved February 12, 2022 from https://docs.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-overview
[7] 2021. *Azure Machine Learning.* Retrieved February 12, 2022 from https://azure.microsoft.com/en-us/services/machine-learning

[8] 2021. *Azure SQL Database serverless.* Retrieved February 12, 2022 from https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview#auto-pausing-and-auto-resuming
[9] 2021. *Azure Synapse SQL architecture.* Retrieved February 12, 2022 from https://docs.microsoft.com/en-us/azure/synapse-analytics/sql/overview-architecture
[10] 2021. *Dynamic Resource Allocation.* Retrieved February 12, 2022 from https://spark.apache.org/docs/3.2.1/job-scheduling.html#dynamic-resource-allocation
[11] 2021. *Get Azure SQL SKU recommendations.* Retrieved February 12, 2022 from https://docs.microsoft.com/en-us/sql/dma/dma-sku-recommend-sql-db?view=sql-server-ver15
[12] 2021. *Predictive scaling for Amazon EC2 Auto Scaling.* Retrieved February 12, 2022 from https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-predictive-scaling.html
[13] 2022. *Amazon Athena.* Retrieved February 12, 2022 from https://aws.amazon.com/athena
[14] 2022. *Amdahl's law.* Retrieved February 12, 2022 from https://en.wikipedia.org/wiki/Amdahl%27s_law
[15] 2022. *Apache Spark.* Retrieved February 12, 2022 from https://spark.apache.org
[16] 2022. *Auto-suspension and Auto-resumption - Snowflake Documentation.* Retrieved February 12, 2022 from https://docs.snowflake.com/en/user-guide/warehouses-overview.html
[17] 2022. *BigQuery.* Retrieved February 12, 2022 from https://cloud.google.com/bigquery
[18] 2022. *Coefficient of variation.* Retrieved February 12, 2022 from https://en.wikipedia.org/wiki/Coefficient_of_variation
[19] 2022. *Data Lake: Microsoft Azure.* Retrieved February 12, 2022 from https://azure.microsoft.com/en-us/solutions/data-lake
[20] 2022. *Multi-cluster Warehouses - Snowflake Documentation.* Retrieved February 12, 2022 from https://docs.snowflake.com/en/user-guide/warehouses-multicluster.html
[21] 2022. *ONNX.* Retrieved February 12, 2022 from https://onnx.ai
[22] 2022. *Permutation feature importance.* Retrieved February 12, 2022 from https://scikit-learn.org/stable/modules/permutation_importance.html
[23] 2022. *sklearn.ensemble.RandomForestRegressor.* Retrieved February 12, 2022 from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
[24] 2022. *Snowflake Data Cloud.* Retrieved February 12, 2022 from https://www.snowflake.com
[25] 2022. *Spark SQL Performance Tests.* Retrieved February 12, 2022 from https://github.com/databricks/spark-sql-perf
[26] 2022. *Synapse Autoscaling.* Retrieved February 12, 2022 from https://docs.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-autoscale
[27] 2022. *TPC-DS Homepage.* Retrieved May 14, 2022 from https://www.tpc.org/tpcds/default5.asp
[28] 2022. *Working with concurrency scaling - Amazon Redshift.* Retrieved February 12, 2022 from https://docs.aws.amazon.com/redshift/latest/dg/concurrency-scaling.html
[29] 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
[30] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14).* USENIX Association, Broomfield, CO, 285–300. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin
[31] Nicolas Bruno, Johnny Debrodt, Chujun Song, and Wei Zheng. 2022. Computation Reuse via Fusion in Amazon Athena. In *2022 IEEE 38th International Conference on Data Engineering (ICDE).* IEEE.
[32] 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. *PVLDB* 1, 2 (2008), 1265–1276.
[33] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data.* 215–226.
[34] 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.* Article 2, 4 pages.
[35] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. 2020. To Tune or Not to Tune? In Search of Optimal Configurations for Data Analytics. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20).* Association for Computing Machinery, New York, NY, USA, 2494–2504.
[36] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11).* USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi11/dominant-resource-fairness-fair-allocation-multiple-resource-types
[37] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *PVLDB* 4, 11 (2011), 1111–1122.
[38] 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 *Cidr*, Vol. 11. 261–272.
[39] Zhiyao Hu, Dongsheng Li, Dongxiang Zhang, and Yixin Chen. 2020. ReLoca: Optimize Resource Allocation for Data-parallel Jobs using Deep Learning. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications.* 1163–1171.
[40] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload Optimization for Cloud Query Engines. In *Proceedings of the ACM Symposium on Cloud Computing.* 416–427.
[41] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation.* 117–134.
[42] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. 2014. MRONLINE: MapReduce online performance tuning. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing.* 165–176.
[43] Abhishek Modi, Kaushik Rajan, Srinivas Thimmaiah, Prakhar Jain, Swinky Mann, Ayushi Agarwal, Ajith Shetty, Shahid K I, Ashit Gosalia, and Partho Sarthi. 2021. New Query Optimization Techniques in the Spark Engine of Azure Synapse. *Proc. VLDB Endow.* 15, 4 (dec 2021), 936–948. https://doi.org/10.14778/3503585.3503601
[44] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.* 301–312.
[45] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. 2015. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *PVLDB* 8, 7 (2015), 726–737.
[46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[47] Anish Pimpley, Shuo Li, Rathijit Sen, Soundararajan Srinivasan, and Alekh Jindal. 2022. Towards Optimal Resource Allocation for Big Data Analytics. In *25th International Conference on Extending Database Technology (EDBT).* 338–350.
[48] 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).
[49] Olga Poppe, Tayo Amuneke, Dalitso Banda, Aritra De, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Deepak Shankargouda, Meina Wang, Alan Au, Carlo Curino, Qun Guo, Alekh Jindal, Ajay Kalhan, Morgan Oslake, Sonia Parchani, Vijay Ramani, Raj Sellappan, Saikat Sen, Sheetal Shrotri, Soundararajan Srinivasan, Ping Xia, Shize Xu, Alicia Yang, and Yiwen Zhu. 2020. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. *PVLDB* 14, 2 (oct 2020), 154–162. https://doi.org/10.14778/3425879.3425886
[50] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. 2016. Perforator: eloquent performance models for resource optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing.* 415–427.
[51] Abhishek Roy, Alekh Jindal, Priyanka Gomatam, Xiating Ouyang, Ashit Gosalia, Nishkam Ravi, Swinky Mann, and Prakhar Jain. 2021. SparkCruise: Workload Optimization in Managed Spark Clusters at Microsoft. *PVLDB* 14, 12 (2021).
[52] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. 2020. AutoToken: Predicting Peak Parallelism for Big Data Analytics at Microsoft. *PVLDB* 13, 12 (2020), 3326–3339.
[53] Rathijit Sen, Abhishek Roy, Alekh Jindal, Rui Fang, Jeff Zheng, Xiaolei Liu, and Ruiping Li. 2021. AutoExecutor: Predictive Parallelism for Spark SQL Queries. *PVLDB* 14, 12 (2021), 2855–2858.
[54] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing.* 1–16.
[55] Lalitha Viswanathan, Alekh Jindal, and Konstantinos Karanasos. 2018. Query and Resource Optimization: Bridging the Gap. In *34th IEEE International Conference on Data Engineering.* 1384–1387.
[56] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. 2018. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Engineering Bulletin* 41, 4 (2018), 39–45.