Predictive Price-Performance Optimization for Serverless Query Processing

Rathijit Sen
Microsoft

Abhishek Roy
Microsoft

Alekh Jindal
Microsoft

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 [2], Google BigQuery [9], Synapse Spark [5], and Synapse SQL [8], 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 or other resource recommendations based on past usage patterns, e.g., SKU recommendations in SQL Server [12]. Others such as Sparklens [18] 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 [7], auto-scaling the resources either at the cluster or pool level as in Synapse Spark [21], RedShift [22], Snowflake [13], etc., or at the query level as in Spark [11] or even Cosmos [24] based on the availability of spare resources. Unfortunately, these reactive approaches take several minutes to react [21] 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 [38]. Therefore, apart from the reactive approaches, we also need predictive resource allocation to provide a good starting point in the first place.

Unfortunately, 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 [30]. Furthermore, changing resources for a query impacts its performance, something which is not well understood by users. To illustrate, Figure 1 shows the performance of TPC-DS query 94 implemented in Spark SQL when using different number of Spark executors, the unit of resources available to queries in Spark. The green curve shows that the performance improves (lower runtime) initially as more executors are added, however it plateaus later on. The labels in red show the total executor occupancy (measured in executor-seconds) over 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 (price-perf for short) optimization at hand.

In this paper, we study predictive price-perf 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 [38], predictive degree of parallelism in SQL Server [26], peak [36], adaptive [23] and optimal [32] resource
allocation in SCOPE [25] jobs, and present an end-to-end framework for predictive price-perf optimization at the query level. We recently demonstrated this system design and concept [37], and in this paper we present more generalized models, detailed architecture, analysis, and results, apart from Spark optimizer extensions that combine both the predictive and the reactive approaches.

Specifically, we make the following key contributions:

1. We discuss current approaches for query-level resource allocation and use production workload insights from Microsoft to motivate why it is important yet difficult to predictively allocate resources at the query-level. Section 2.

2. We present a parametric model to analyze the price-perf trade-off and predictively pick the point of operation for a query. We further discuss the impact of the total number of cores, apart from the executor count. We describe how to train the price-perf model parameters using different machine learning models and discuss the feature importance. Section 3

3. We describe, AutoExecutor, an system that integrates predictive price-perf optimization with the Spark query optimizer, and discuss various aspects including the model format, model lookup, optimizer extensions, and resource deallocation by combining both the predictive and the reactive resource allocation. Section 4

4. Finally, we show a detailed set of experiments on predictive price-perf optimizations in Spark SQL using the TPC-DS workload. We discuss prediction accuracy, picking the point of operation, relationship with dynamic allocation, handling workload changes, and the overheads involved. Our results show that on TPC-DS, AutoExecutor can save executor occupancy by 48% compared to dynamic allocation's reactive behavior. Section 5.

2 QUERY RESOURCE ALLOCATION

In this section, we discuss various query-level resource allocation approaches. We focus our discussion on the Spark query processing engine and 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.

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, that we denote by \( \text{AUC} = \int_a^b n_s ds \), and that impacts the total resource reservation. \( \text{AUC} \) is the Area Under the (skyline) Curve in a timeline plot of \( n_s \) vs \( s \) over the lifetime of the query execution.

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

2.1 Why Per-query Resource Allocation?

The first question is why does per-query resource allocation matter. We analyze a large subset of daily production Spark workloads at Microsoft consisting of 90, 224 applications and 840, 278 queries across 3, 245 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 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 disaggregated storage service in Azure Synapse [10]. Thus, it is more practical to allocate resources efficiently at per-query level within each Spark application.
2.2 The Default Behavior
Let us now look at the default behavior when allocating the executor counts for each Spark SQL query. Our production Spark workloads at Microsoft show that 59% of the Spark applications have dynamic allocation 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 ideal, with unrealistic minimum and maximum values set for the executor counts.

2.3 Reactive Approaches
We now discuss the major reactive approaches to deal with the above default behavior.

First of all, Spark dynamic allocation [11] 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 [18] 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 for future instances of the same query. Users can construct the skyline of executor count and get recommendations for better executor count allocations. Still, the approach is limited to improving the resource allocation for the same query and requires analysis for every such query, which may not be possible for large production workloads. Other approaches such as Tuneful [27], Reloca [28], and Perforator [34] 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 [21], 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 [6, 7, 14, 33]. 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
Let us now see why 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 two 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.

3 PRICE-PERF MODEL
In this section, we present our approach to price-perf model (PPM for short). We build on top of prior work on predicting optimal resources for SCOPE jobs at Microsoft [32], and extend it to a more
generalized framework for price-perf optimization. We discuss this framework below.

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:

\[
t(n) = f(n, \text{[scalar parameters]})
\]

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

\[
g : \text{query characteristics} \mapsto \text{[scalar parameters]}
\]

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.

While selecting candidate functions for \( f \), we impose a condition of monotonicity, similar as in the prior work [32]. This condition means that \( t(n) \) should be monotonically non-increasing with \( n \). This is consistent with user expectations that the run time of a characteristic curve from the prior work [32], which uses a power-function of our resource model decisions and with forecasting the future expectations with simple monotonic models helps with explainability monotonically non-increasing; (4) being consistent with user expectations that the run-time estimates from Sparklens, that we use to 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.

3.2 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 \( \text{AE\_PL} \) and \( \text{AE\_AL} \) 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 \( \text{AE\_AL} \) is a better fit to Sparklens estimates for \( n < 32 \) while \( \text{AE\_PL} \) is a better fit beyond that. Thus, one can obtain an error of 7% or less for the full range of \( n \) by using the models over different ranges of \( n \). Interestingly, we see that although \( \text{AE\_AL} \) fits Sparklens estimates better at lower values of \( n \), it does not translate into better prediction accuracy than \( \text{AE\_PL} \) when compared to actual query run times in a number of cases, as we will show in Section 5.

PPM model is thus formulated as:

\[
t(n) = \max(b \times n^a, m)
\]

This model has three query-specific parameters, \( a, b, \) and \( m \), that the ML model will learn and predict. We abbreviate this model by \( \text{AE\_PL} \) in the rest of this work.

\textbf{Amdahl’s Law:} This is inspired by the well-known Amdahl’s Law model for computation speedup with increase in resources used for the computations [3]. 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:

\[
t(n) = s + \frac{p}{n}
\]

This model has two query-specific parameters, \( s \) and \( p \), that the ML model will learn and predict. We abbreviate this model by \( \text{AE\_AL} \) in the rest of this work.

Figure 4: PPM model errors for different executor counts for two models, \( \text{AE\_PL} \) and \( \text{AE\_AL} \), when fit on Sparklens estimates of the PPM over all queries of TPC-DS SF=100.
Table 1: Configurations used for observing impact of total cores on query run time.

| Cores/Executor (e_c) | Executors (n) | Total Cores (k) |
|----------------------|---------------|-----------------|
| 2                    | 3             | 6               |
|                      | 16            | 32              |
| 4                    | 1             | 4               |
|                      | 3             | 12              |
|                      | 4             | 16              |
|                      | 8             | 32              |
|                      | 16            | 64              |
|                      | 32            | 128             |
|                      | 48            | 192             |
| 6                    | 3             | 18              |
|                      | 16            | 96              |
| 8                    | 3             | 24              |
|                      | 16            | 128             |

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 × 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.

To explore this idea, we evaluate several configurations with different values of n and e_c as shown in Table 1. Figure 5 shows actual run times of two example queries from TPC-DS, SF=100, for these configurations. We group configurations into series by the e_c values while the x-axes show the values for k. Figure 5a shows that all the points line up almost on top of the series for e_c = 4 suggesting that the value of k matters more to overall performance rather than how it is factorized into n and e_c. Although the points in Figure 5b are not fully aligned, the distances from the e_c = 4 trend line are small.

Figure 5c shows the distributions of the relative distances from the linearly-interpolated e_c = 4 values for configurations with e_c ≠ 4, for the same value of k. According to Table 1, there are six configurations with e_c ≠ 4 for each query, and Figure 5c includes characteristics for both query plan as well as the inputs to the query and is motivated by our observation that the optimal executor count depends on both of these aspects (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 over 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 the AE_PL model, m is the minimum run time over all the different configurations. The power-law portion of the PPM can be transformed into logarithmic space:

\[
\log(t(n)) = \log(b) + n \times \log(a) \tag{5}
\]

\(\log(a)\) 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 \(n > n_m\).

For the AE_AL model, s and p can be determined by fitting a linear regression model to \(t(n)\) as a function of \(\frac{n}{n_m}\).

Table 2 shows the features used for the parameter model. This includes characteristics for both query plan as well as the 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...
number of input data sources used

estimated total number of bytes of

description

maximum depth of query plan

total number of operators in the

(c) error distribution

Figure 5: (a) and (b) show run times for two example TPC-DS queries, SF=100, with different number of executors, _n_, and cores per executor, _e_c_ (also see Table 1). Configurations with the same _e_c_ value belong to the same series. The legend entries show the different series (different _e_c_ values). (c) shows distributions of the relative errors in run time estimation for configs with _e_c_ ≠ 4, with estimation using linear interpolation from the run times for _e_c_ = 4, for all queries of TPC-DS SF=100.

Table 2: 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) |
| ∑ 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                            |
| ∑ Input bytes                  | Estimated total number of bytes of input data used by the query            |
| ∑ Rows processed               | Estimated total number of rows processed by all operators in the query     |

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 feature importances to the model 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×_c_t_ data points where _c_t_ is the number of training configurations for each query. If there are _c_t_ candidate configurations for each test query, then it would score the model _c_t_ 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-perf optimization with the Spark query engine [4]. Traditionally, the 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 6 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 2. To collect this information, we use Peregrine [29] and SparkCruise [35] 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 [18]. 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 6. 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

Currently, the *AutoExecutor* pipeline uses Scikit-learn Python library to train models. However, the Spark optimizer code runs inside Java Virtual Machine (JVM) 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 [15]. 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. 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

Once the ONNX model is produced, we can leverage typical model management libraries and infrastructure, such as Azure Machine Learning or MLflow. Thereafter, *AutoExecutor* introduces prediction-based optimizations in the Spark query optimizer and applies a series of five steps, as illustrated in Figure 6, 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. Then, we featureize 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-perf 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.

---

Figure 6: The *AutoExecutor* system design illustrating the prediction-based resource optimization in Spark.

Figure 7: Combining predictive and reactive resource allocation for individual queries in an interactive Spark application.
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 deallocate 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 [23]. 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 7 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 executor count of 22 and automatically requests the predicted executor count. 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 experiment evaluation of AutoExecutor. Our goal is to answer several key questions, including how good are the models, what is the impact of price-perf 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. Below we first described out setup before discussing each of these questions.

5.1 Setup

Our testbed consists of 103 TPC-DS queries (99 queries + variants) [20], for two scale factors SF=10 and 100, 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 ($c_e = 4$) 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 $\pm 1.5\times$ the inter-quartile range. The 'Actual' series shown in the figures correspond to this averaged run time data.

![Figure 8: Comparing Sparklens estimates, PPM predictions before query execution using AE_PL and AE_AL models, and the actual runtimes for TPC-DS query 94, SF=100.](image)

![Figure 9: 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 $\pm 1$ standard deviation across the $5 \times 10 = 50$ testing folds.](image)
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. Of course, the testing dataset only uses runtime
from actual runs.

For each query, we obtain Sparklens estimates for the application
time by running the tool on the executor logs after a single run of
that query with \( n = 16 \). We show these estimates as the series ‘S’
in the figures.

For evaluating how well the model predictions generalize across
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 Am-
dahl’s Law, that we show as AE.PL and AE.AL series in the figures.

5.2 Time Prediction

We now evaluate how well the run times can be predicted for
different values of \( n \). Figure 8 shows example time predictions for
a query and compares it with the actual runtimes for that query.
Note that the training dataset for the PPM here did not include this
query. We also show Sparklens estimates for comparison. However,
to get the Sparklens estimates, we need to execute the query once,
in this case with \( n = 16 \). We fix \( e_c = 4 \) for all configurations
considered for this example. Although the model predictions as
well as the Sparklens estimates differ from the actual run times at
small executor counts, e.g., at \( n = 1 \), the PPMs become similar at
higher executor counts. Although there is a small difference in the
actual execution times, the shapes of the curves are similar.

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.

\[
E(n) = \frac{\sum_q \left| t_q(n) - \hat{t}_q(n) \right|}{\sum_q t_q(n)}
\]  

(6)

Thus, \( E(n) \) measures 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 \( n \).

Figures 9a and 9b show average \( E(n) \) for the training and testing
datasets 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. The errors thus look to
be dominated more by bias rather than by variance, and the models
are not over-fitted. The relatively larger absolute 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 much closer to Sparklens
estimation errors. This is because we augment the training data
for the PPM model with the Sparklens estimates. The training
(fit) errors show that the AE_AL fit is better than that for AE.PL
for \( n < 32 \), but slightly worse beyond, which is in line with our
observations from Figure 4. However, from the testing (prediction)
errors we see that the gap is reduced and AE.PL does better at
\( n = 1, 48 \). The average absolute difference in \( E(n) \) values from
Sparklens was quite small: 0.079 for AE.PL and 0.094 for AE_AL
on the 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. For
these experiments, we 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 time \( t_{\min} \) does not exceed a threshold \( H \), i.e., \( \frac{t(n)}{t_{\min}} \leq H \). Figure 10a shows the slowdowns using the (interpolated) actual
run times for the selected configurations for different values of
\( H \), while Figure 10b shows the corresponding \( n \). We average over
the queries in the test datasets for the 10-repeated 5-fold cross
validations and the error bars correspond to \( \pm 1 \) 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

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figures/figure10.png}
\caption{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 \( \pm 1 \) standard deviation across the \( 5 \times 10 = 50 \) testing folds.}
\end{figure}
over \( t_{\text{min}} \). We find that with the selected configurations, there is a resulting additional slowdown of 5.4% for Sparklens, 5.5% for AE\textsubscript{PL}, and 8.9% for AE\textsubscript{AL}. The average values for \( n \) are 24 for Actual and 32.9, 21.5, and 48 respectively for Sparklens, AE\textsubscript{PL}, and AE\textsubscript{AL}, thus reflecting a significant savings opportunity. AE\textsubscript{PL} improves over both Sparklens and AE\textsubscript{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\textsubscript{AL} always select the maximum value of \( n = 48 \) due to the lack of a saturation constraint on the model unlike AE\textsubscript{PL}.

As discussed in Section 2.2, users often run jobs with static (same settings for all jobs) and small values of \( n \). Using the AE\textsubscript{PL} or AE\textsubscript{AL} models, we can automatically choose a configuration that optimizes for \( H = 1 \) as above, that provides significant speedups over static , small values of \( n \). The above selected configurations provided an average speedup of 69–70% for static \( n = 3 \) (12 cores) and 12.6–13.8% for \( n = 8 \) (32 cores). Speedups over static \( n = 2 \) (8 cores) would be higher (expected 2.6–2.7\times 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\textsubscript{PL} for \( H = 1.05, 1.1, 1.2, 1.5, 2 \) were 1.06, 1.06, 1.07, 1.12, 1.28 for AE\textsubscript{PL} (with average \( n = 19.7, 18.2, 15.7, 10.7, 6.2 \)), but 1.04, 1.08, 1.16, 1.38, 1.67 (with average \( n = 15.3, 12, 8.2, 4.9, 3.2 \)) for Actual. Overall, AE\textsubscript{AL} tends to significantly overestimate \( n \). While AE\textsubscript{PL} realizes only a part of the full savings potential, it has a similar impact as that of Sparklens but it 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 curves of Figures 1 and 8 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 the PPM figures, 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. Our approach is to normalize both \( n \) and \( t(n) \) using range-scaling functions \( u : n \mapsto [0, 1] \) and \( v : t(n) \mapsto [0, 1] \), and then compute the slope at each point \( u(n) \) of the normalized PPM as follows.

\[
 u(n) = \frac{n - \min(n)}{\max(n) - \min(n)}
\]

\[
v(t(n)) = \frac{t(n) - \min(t(n))}{\max(t(n)) - \min(t(n))}
\]

\[
slope(u(n)) = \frac{v(t(n+1)) - v(t(n))}{u(n) - u(n+1)}
\]

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 \( \text{slope}(u(n)) \geq 1 \) and \( \text{slope}(u(n + 1)) \leq 1 \).

Figure 11 shows the distribution of \( L \) over all queries of TPC-DS, SF=100. The vast majority of queries have \( L = 8 \); only 13 of 103 queries have \( L < 8 \) for Actual. 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. Interestingly, AE\textsubscript{AL} always selected \( L = 7 \) for these queries. AE\textsubscript{PL} selected 8, 9, or 10 for \( L \). Overall, model-predicted elbow points were close to the actual values in most cases.

5.4 Cost savings compared to static and dynamic allocation

Figure 12 shows the executor allocation skyline for an example query when run using four policies: dynamic allocation (DA) with \( n \) restricted to the range of \([1, 48]\); static allocation (SA), that is all executors requested upfront during job submission, once with \( n = 48 \) and once with \( n = 25 \); a request of 25 total executors made during the AutoExecutor optimizer rule (Rule\textsubscript{25}) for a run that started with \( n = 5 \). The 25 executor count was predicted by AE\textsubscript{PL} for this query in one of the 5-fold validation experiments with optimization objective of \( H = 1.05 \) (see Section 5.3).

We note that while the run times for SA\textsubscript{48} and SA\textsubscript{25} were close, the latter substantially reduced total \( n \) (48\rightarrow25) and \( \text{AUC} \) (1904\rightarrow1022). Similar observations hold between DA\textsubscript{1,48} and
We now evaluate how well the models perform when the test dataset consists of all TPC-DS queries for that scale factor, which is SF=10 for Figure 14a and SF=100 for Figure 14b. In each case, the training dataset is constructed from the other scale factor, that is SF=100 and 10 respectively for the two figures. For Sparklens, we show two estimates for reference, one from running queries at SF=10 and the other from running queries at SF=100, both with n = 16.

Overall, we see similar trends as when changing the query templates for the testing dataset (see Figure 9). In particular, we see larger errors at small values of n for AE_PL as well as AE_AL. The error ranges are also comparable for the two scenarios. Interestingly, sometimes the model predictions improve over the Sparklens estimates from the training dataset, e.g., AE_PL vs S_10 for testing SF=100 and n ≥ 16. We also notice a significant difference in the average errors between S_10 and S_100 for both testing datasets for several n, e.g., S_10 vs S_100 for n = 8 and testing SF=10, for n = 16 and testing SF=100, etc. This can be understood in the light of the fact that while query behaviors change with the amount of data processed, Sparklens does not account for the change in data sizes while generating timing estimates.

### 5.6 Overheads

We now discuss various training and scoring 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 [19] for the Random Forest model. The pickled file size on disk when trained over all 130 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 [17]. Figure 15 lists the top 10 features in decreasing order of the sum of average importance scores for the model predictions on the testing datasets. The most important features are the estimated total input bytes and rows processed, followed by maximum depth of the query plan, number of operators, then specific operators such as Project and Filter. Some other operators, not shown in Figure 15, appeared much less frequently or changed less in value across queries in our dataset and got low importance scores.

To further assess the impact of these features, we performed an ablation study by constructing three reduced feature sets from the original feature set F0: (1) F1 consisting of the top six features in Figure 15, (2) F2 consisting of the top two features (affected by input data size), and (3) F3 = F1 − F2, that is, the top four plan features. Errors $E(n)$ for F1 were close to those for F0, but those for F3 were larger, followed by F2. For example, at $n = 8$, $E(n)$ for F0, F1, F2, F3 were 0.27, 0.26, 0.35, 0.31 for AE_PL and 0.24, 0.24, 0.3, 0.27 for AE_AL. These results, as well as the feature rankings, reinforce our view that both input sizes and plan features together impact query run times and their predictions.

6 CONCLUSIONS

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. 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 48% of executor occupancy compared to dynamic allocation.

REFERENCES

[1] Add hooks and extension points to Spark. https://issues.apache.org/jira/browse/SPARK-18127.
[2] Amazon Athena. https://aws.amazon.com/athena.
[3] Amdahl’s law. https://en.wikipedia.org/wiki/Amdahl%27s_law.
[4] Apache Spark. https://spark.apache.org.
[5] Apache Spark in Azure Synapse Analytics. https://docs.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-overview.
[6] Auto-suspension and Auto-resumption - Snowflake Documentation. https://docs.snowflake.com/en/user-guide/warehouses-overview.html.
[7] Azure SQL Database serverless. https://docs.microsoft.com/en-us/azure/sql/database/serverless-tier-overview#auto-pausing-and-auto-resuming.
[8] Azure Synapse SQL architecture. https://docs.microsoft.com/en-us/azure/synapse-analytics/sql/overview-architecture.
[9] BigQuery. https://cloud.google.com/bigquery.
[10] Data Lake: Microsoft Azure. https://azure.microsoft.com/en-us/solutions/data-lake.
[11] Dynamic resource allocation. https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation.
[12] Get Azure SQL SKU recommendations. https://docs.microsoft.com/en-us/sql/dma/dma-sku-recommend-sql-db?view=sql-server-ver15.
Multi-cluster Warehouses - Snowflake Documentation. https://docs.snowflake.com/en/user-guide/warehouses-multicluster.html.

New – predictive scaling for EC2, powered by machine learning. https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/.

ONNX. https://onnx.ai.

Part 3: Cost efficient executor configuration for Apache Spark: https://medium.com/expedia-group-tech/part-3-efficient-executor-configuration-for-apache-spark-b4602929262.

Permutation feature importance. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html.

Qubole Sparklens tool for performance tuning Apache Spark. https://github.com/qubole/sparklens. v0.3.2.

sklearn.ensemble.RandomForestRegressor. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html.

Synapse Autoscaling. https://docs.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-autoscale.

Working with concurrency scaling - Amazon Redshift. https://docs.aws.amazon.com/redshift/latest/dg/concurrency-scaling.html.

M. Bag, A. Jindal, and H. Patel. Towards plan-aware resource allocation in serverless query processing. In 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20). USENIX Association, 2020.

E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pages 285–300, Broomfield, CO, 2014. USENIX Association.

R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Part 3: Cost efficient executor configuration for Apache Spark. https://medium.com/expedia-group-tech/part-3-efficient-executor-configuration-for-apache-spark-b4602929262.

Z. Hu, D. Li, D. Zhang, and Y. Chen. Reloca: Optimize resource allocation for data-parallel jobs using deep learning. In IEEE INFOCOM 2020 - IEEE Conference on Computer Communications, pages 1163–1171, 2020.

A. Jindal, H. Patel, A. Roy, S. Qiao, Z. Yin, R. Sen, and S. Krishnan. Peregrine: Workload optimization for cloud query engines. In Proceedings of the ACM Symposium on Cloud Computing, pages 416–427, 2019.

S. A. Jyothi, C. Curino, I. Menache, S. M. Narayananmurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J.ulkarni, et al. Morphues: Towards automated SLOs for enterprise clusters. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, pages 117–134, 2016.

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. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12:2825–2830, 2011.

A. Pimpley, S. Li, A. Srivastava, V. Rohra, Y. Zhu, S. Srinivasan, A. Jindal, H. Patel, S. Qiao, and R. Sen. Optimal resource allocation for serverless queries. arXiv preprint arXiv:2107.08594, 2021.

O. Poppe, T. Annemuele, D. Banda, A. De, A. Green, M. Knoertzer, E. Nosakhare, K. Rajendran, D. Shankargouda, M. Wang, A. Au, C. Curino, Q. Guo, A. Jindal, A. Kalhan, M. Ostlake, S. Parchani, V. Ramani, R. Sellappan, S. Sen, S. Shroti, S. Srinivasan, P. Xia, S. Xu, A. Yang, and Y. Zhu. SeaGull: An infrastructure for load prediction and optimized resource allocation. PVLDB, 14(2):154–162, 2020.

K. Rajan, D. Kakadia, C. Curino, and S. Krishnan. Perforator: eloquent performance models for resource optimization. In Proceedings of the Seventh ACM Symposium on Cloud Computing, pages 415–427, 2016.

A. Roy, A. Jindal, P. Gomadam, X. Quyang, A. Gosalia, N. Ravi, S. Mann, and P. Jain. SparkCruise: Workload optimization in managed spark clusters at microsoft. PVLDB, 14(12), 2021.

R. Sen, A. Jindal, H. Patel, and S. Qiao. AutoToken: Predicting peak parallelism for Big Data analytics at Microsoft. PVLDB, 13(12):3326–3339, 2020.

R. Sen, A. Roy, A. Jindal, R. Fang, J. Zheng, X. Liu, and R. Li. AutoExecutor: Predictive parallelism for Spark SQL queries. PVLDB, 14(12):2855–2858, 2021.

L. Vinwathanath, A. Jindal, and K. Karanasos. Query and resource optimization: Bridging the gap. In 34th IEEE International Conference on Data Engineering, pages 1384–1387, 2018.