AutoExecutor: Predictive Parallelism for Spark SQL Queries

Rathijit Sen
Microsoft
Madison, United States
rathijit.sen@microsoft.com

Abhishek Roy
Microsoft
Redmond, United States
abhishek.roy@microsoft.com

Alekh Jindal
Microsoft
Redmond, United States
alekh.jindal@microsoft.com

Rui Fang
Microsoft
Beijing, China
rufan@microsoft.com

Jeff Zheng
Microsoft
Beijing, China
jezheng@microsoft.com

Xiaolei Liu
Microsoft
Beijing, China
xiaolel@microsoft.com

Ruiping Li
Microsoft
Sunnyvale, United States
ruiping.li@microsoft.com

ABSTRACT

Right-sizing resources for query execution is important for cost-efficient performance, but estimating how performance is affected by resource allocations, upfront, before query execution is difficult. We demonstrate *AutoExecutor*, a predictive system that uses machine learning models to predict query run times as a function of the number of allocated executors, that limits the maximum allowed parallelism, for Spark SQL queries running on Azure Synapse.

PVLDB Reference Format:
Rathijit Sen, Abhishek Roy, Alekh Jindal, Rui Fang, Jeff Zheng, Xiaolei Liu, and Ruiping Li. AutoExecutor: Predictive Parallelism for Spark SQL Queries. PVLDB, 14(12): 2855 - 2858, 2021.
doi:10.14778/3476311.3476362

1 INTRODUCTION

Resource optimization in cloud-based analytical query processing is a challenging problem. This is because while on one hand modern cloud analytics platforms make it possible to allocate resources in a fine-grained per-query basis (e.g., Azure Data Lake [9], Azure Synapse [7], AWS Athena [6], Google BigQuery [8]), on the other hand, it is still very difficult to anticipate the relationship between resource allocation and query performance. Sub-optimal resource allocation can lead to poor performance, higher costs, and inefficient cluster utilization. Given the complexity of the problem, prior work has leveraged machine learning to build models from the past seen workloads [13, 16, 18], especially with the presence of massive workloads on the cloud [14]. Unfortunately, most of the early efforts considered queries as black box without considering their shape and characteristics, and thus risking either overfitting or overgeneralization. Our recent work on SQL Server [12] and Cosmos workloads [20] overcomes this problem by leveraging query plan characteristics to learn more accurate and robust resource models that improve not just the query performance, but also the data movement, cluster throughout, and query wait times. We further introduced an explainable approach that visually represents a query's run time as a function of given resources [17], called the Performance Characteristic Curve (PCC), to make resource allocation decisions even more transparent to the users.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476362

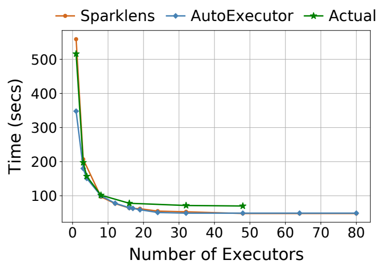


Figure 1: Performance Characteristic Curve for an example TPC-DS query (q94) as a function of the number of allocated executors from Actual runs, Sparklens estimate (post-execution), and AutoExecutor prediction (pre-execution).

In this paper, we continue our investigation on resource optimization beyond SQL Server and Cosmos, and apply it to Spark workloads. Our goal is to model the resource-performance behavior of Spark SQL queries and put that in context with two newer trends in the Spark community, namely the dynamic resource allocation [10] and post-execution analysis tools such as Sparklens [4]. Specifically, given a Spark SQL query, we want to predict the optimal degree of parallelism, or the number of *executors*, which we found to be the most sensitive resource parameter for a given Spark environment such as Azure HDInsight [5] or Azure Synapse [3]. Our predictive approach complements the dynamic allocation feature [10] in Spark, that adapts the executor count during the course of query processing, since relying solely on dynamic allocation leads to problems such as a time lag to build up the resources from the start of the query execution, resource overshoot due to the exponential resource build up, and several auto-scale requests during the resource build up. Thus, in addition to the dynamic behavior, it is still crucial to have a good starting point for resource allocation.

Figure 1 shows three PCCs for an example TPC-DS query running on a Spark pool in Azure Synapse [3]. The 'Actual' PCC is obtained by running the query with different executor counts. The 'Sparklens' PCC is obtained by running Sparklens [4], a popular tool that estimates the PCC from a single run of a query, in this case run with 16 executors. Sparklens can only simulate PCCs for the exact same query, whereas production workloads often contain recurring queries [15] that get executed over newer datasets or with different parameters and that could have very different PCCs over time.

Furthermore, we want to predict the PCC upfront, *before query execution*, so that users can choose the optimal configuration for that query according to their cost-performance optimization objective. Therefore, we need a predictive approach to resource allocation at the beginning of a Spark query. The third PCC in Figure 1 is from our system, *AutoExecutor*, that we propose to demonstrate. AutoExecutor is a predictive resource allocator for Spark SQL queries. It predicts the PCC for an incoming query *before execution* for different executor counts (thus determining the parallelism achieved by the query). For this example, it predicted the PCC well, although it had not seen the same query beforehand.

AutoExecutor solves multiple problems at the same time, including relieving the user from analyzing query executions and tuning resources, adapting to changes in workload characteristics, especially the inputs sizes in recurring queries, improving query performance by more aggressive resource allocation upfront when more resource-intensive parts of the query plans are executed anyways, freeing up redundant resources for other applications in the same cluster to make progress, reducing auto-scaling overheads by helping make few scaling up or down requests, helping analyze the VM requirements by looking at the predictions for each query in the workload, even forecasting the changing resource requirements based on changes in workload, and better explaining the resource-performance trade-off via a feature-based approach.

2 AUTOEXECUTOR OVERVIEW

We now describe how we model the PCC and how we built an integrated research prototype on Azure Synapse Spark pools [3].

2.1 PCC model

AutoExecutor leverages the PCC model [17] with the assumption that query run time decreases monotonically with increasing parallelism, i.e., they have a power-law relationship. While this broadly holds true for Spark, there could be cases with specific query characteristics or other query processors where run time may increase with parallelism. However, such cases are typically not interesting from a cost-efficient performance perspective, neither for the customer nor for the service provider. Similar to Sparklens [4] we aim to produce a PCC, though predictively at compile-time, that flattens out beyond a certain parallelism depending on the given query. To model the flattening of the curve, we extend our prior PCC model [17] by imposing a constraint on the minimum value, m, of the predicted run time, $t_p(n)$, for n executors as follows.

$$t_p(n) = max(b \times n^a, m) \tag{1}$$

where a, b, m are scalar constants that depend on the given query.

Our experiments with TPC-DS indicate that the PCC model may also be used with the total number of executor cores, $c = e_c \times n$, where e_c is the number of cores per executor and n is the number of executors. Figure 2 shows the PCC from run times of a TPC-DS query, obtained by varying e_c and n, with the x-axis showing the total executor cores, c. For example, the points at $c = 32$ are obtained from $(n, e_c) = (8, 4)$ and $(16, 2)$. In this example, the points from the different e_c series (legend entries) line up well for the same value of c on the curve. While there may be cases where a particular factorization of c into (n, e_c) may be advantageous than others,

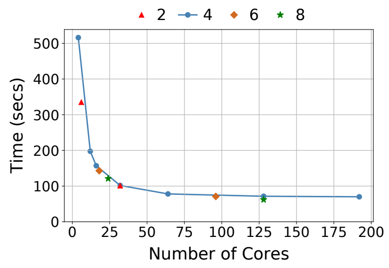


Figure 2: PCC (actual runs) for an example query (q94) as a function of the number of total executor cores (c), with different numbers of cores per executor (e_c, legend entries).

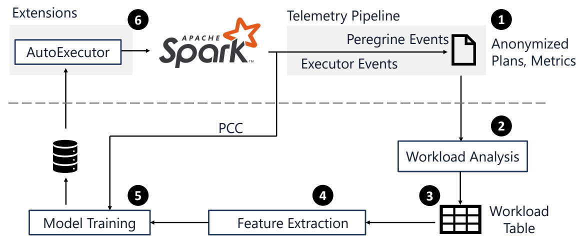


Figure 3: AutoExecutor System Architecture.

modeling the PCC as a function of c is a good first-order approximation. It also simplifies the modeling considerably by reducing the configuration space from two dimensions to one dimension. To model this, we can simply replace n with c in Equation 1.

2.2 Spark Integration

AutoExecutor learns how to correlate PCCs with query characteristics by observing behavior from past executions. We automate this learning process by using machine learning models that map the characteristics of a Spark SQL query to PCC parameters (a, b, m).

AutoExecutor uses Peregrine workload optimization framework for Spark [14, 19]. Figure 3 shows an overview of the system architecture. Operations above the dashed line happen online whereas the rest are executed offline. Step ❶ collects anonymized plans and metrics. Step ❷ analyzes the query plan telemetry and Step ❸ creates a denormalized workload table. Step ❹ extracts features for the PCC model from the workload table. We currently use operator counts, total operators, estimated total input cardinality and total input bytes, number of input sources, and plan depth as features for the model. These are available at query compilation and optimization time. We also need PCC parameters as data labels for the training. These can be obtained by fitting Equation 1, either to actual query run times with different executor counts, or to estimates from simulators such as Sparklens. Step ❺ trains the model and stores it in a database in ONNX format. When a new query is submitted to the system, Step ❻ injects an additional Spark optimizer rule to look up the database for the ONNX model, scores it using feature values for the query, and determines the optimal executor count from the predicted PCC for running the query.

3 EVALUATION

To evaluate the accuracy of AutoExecutor, we performed 5-fold cross validations (with train/test splits of 80%/20%), repeated 10

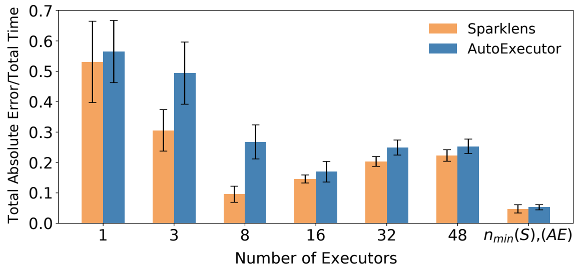


Figure 4: Sum of absolute errors relative to the sum of run times of test queries with different executor counts, for Sparklens (S) and AutoExecutor (AE) predictions. Error bars show ±1 standard deviation with repeated cross-validation.

times, over TPC-DS (scale factor = 100) queries and with $e_c = 4$. We obtained Sparklens estimates for each query from the event logs of a single run of the query *after* running it with $n = 16$. For AutoExecutor, we used a Random Forest model and used parameters from the fitted PCC model (Equation 1) on Sparklens estimates on the training queries as labels for model training. To obtain ground truth values for each query for different values of n, we took average application times over several runs after discarding outliers.

Figure 4 shows total errors compared to total actual run times (ground truth values) over all test queries belonging to that validation fold, for two scenarios. (1) *Time prediction*: for n =1, 3, 8, 16, 32, 48, the errors are the absolute differences of Sparklens (S) and AutoExecutor (AE) time predictions from actual run times for that n. (2) *Configuration selection*: n_{min} is the minimum n estimated from Sparklens and AutoExecutor predictions to have the lowest run time for the query over the above set of values for n, and the errors are the differences of the actual run times at $n_{min}(S)$, $n_{min}(AE)$ from the actual minimum run times for the query.

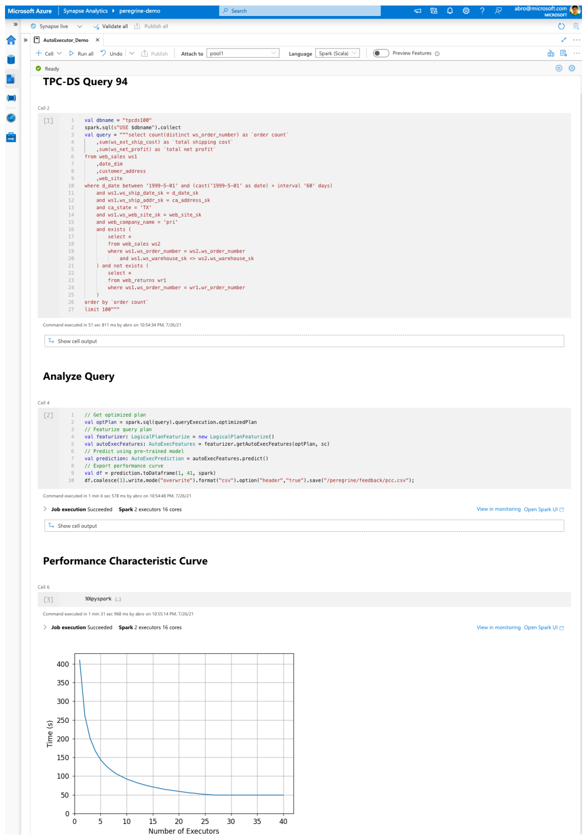
Time prediction errors were somewhat higher for AutoExecutor than for Sparklens, but in contrast to Sparklens, AutoExecutor predictions were made *before* running the test queries. The errors were ~25% or less on average for $n \geq 16$, but larger for smaller values of n. Approximating the PCC with a different function for low n may reduce these errors. Average values for $n_{min}(S)$, $n_{min}(AE)$, and actual n_{min} were 33, 28, and 24 respectively, with average timing sub-optimality of ~5% from running with $n_{min}(S)$ and $n_{min}(AE)$.

4 DEMONSTRATION

We will run the demonstration on Azure Synapse Spark [3] with preloaded TPC-DS dataset. We will encourage the audience to modify the queries or to explore different executor selection scenarios.

4.1 Scenario 1: visualize the resource-performance trade-off

The relationship between resources and performance is hard to anticipate, and yet it is often important for both users and administrators to understand it. Synapse customers regularly use tools like Sparklens to analyze a Spark SQL query post-execution and estimate its performance for other executor counts. Likewise, internal Cosmos users at Microsoft have a similar tool built into Scope Studio, the Visual Studio environment for SCOPE [11], to estimate performance for other token counts (the unit of parallelism for SCOPE jobs). In our demonstration, we will predict and show the PCC for Spark SQL queries. The audience will be encouraged to write their own queries over TPC-DS tables. AutoExecutor will use the learned models to predict the PCC parameters for each of those queries. With this information, the users can select the optimal executor count within their cost budget. This scenario is implemented in an interactive notebook as shown in Figure 5.



TPC-DS Query 94

Cell 2

```
[1]
val dbname = "tpcds100"
spark.sql(s"USE $dbname").collect
val query = """select count(distinct ws_order_number) as `order count`
    ,sum(ws_ext_ship_cost) as `total shipping cost`
    ,sum(ws_net_profit) as `total net profit`
from web_sales ws1
    ,date_dim
    ,customer_address
    ,web_site
where d_date between '1999-5-01' and (cast('1999-5-01' as date) + interval '60' days)
    and ws1.ws_ship_date_sk = d_date_sk
    and ws1.ws_ship_addr_sk = ca_address_sk
    and ca_state = 'TX'
    and ws1.ws_web_site_sk = web_site_sk
    and web_company_name = 'pri'
    and exists (
        select *
        from web_sales ws2
        where ws1.ws_order_number = ws2.ws_order_number
            and ws1.ws_warehouse_sk <> ws2.ws_warehouse_sk
    ) and not exists (
        select *
        from web_returns wr1
        where ws1.ws_order_number = wr1.wr_order_number
    )
order by `order count`
limit 100"""
```

Command executed in 51 sec 811 ms by abro on 10:54:34 PM, 7/26/21

Show cell output

Analyze Query

Cell 4

```
[2]
// Get optimized plan
val optPlan = spark.sql(query).queryExecution.optimizedPlan
// Featurize query plan
val featurizer: LogicalPlanFeaturize = new LogicalPlanFeaturize()
val autoExecFeatures: AutoExecFeatures = featurizer.getAutoExecFeatures(optPlan, sc)
// Predict using pre-trained model
val prediction: AutoExecPrediction = autoExecFeatures.predict()
// Export performance curve
val df = prediction.toDataframe(1, 41, spark)
df.coalesce(1).write.mode("overwrite").format("csv").option("header","true").save("/peregrine/feedback/pcc.csv");
```

Command executed in 1 min 6 sec 578 ms by abro on 10:54:48 PM, 7/26/21

Job execution Succeeded Spark 2 executors 16 cores View in monitoring Open Spark UI

Show cell output

Performance Characteristic Curve

Cell 6

```
[3]
%%pyspark [...]
```

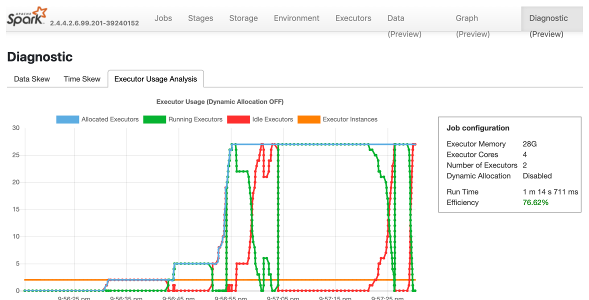
Command executed in 1 min 31 sec 968 ms by abro on 10:55:14 PM, 7/26/21

Job execution Succeeded Spark 2 executors 16 cores View in monitoring Open Spark UI

Figure 5: Visualizing performance-resource trade-off in an Azure Synapse Notebook.

4.2 Scenario 2: automatic selection of an important resource parameter

Spark provides users with high-level API such as Dataframes and Koalas [2] that hide the complexities of distributed data processing. However, given that different queries may work well with different executor counts, it is tedious for users to identify and configure them on a per-query basis. Therefore, in this scenario, we demonstrate how AutoExecutor can automate setting the executor count, an important resource parameter, for each incoming query. AutoExecutor does this by plugging an additional optimizer rule into the Spark optimizer using the Spark extensions API [1]. This new optimizer rule loads the ONNX model, predicts the PCC parameters using compile-time features from the query plan, and requests additional number of executors as required — all automatically during query optimization. As a result, users get a handsfree job submission experience where they no longer have to manually select and configure the number of executors for every query. We will



Job configuration

Executor Memory	28G
Executor Cores	4
Number of Executors	2
Dynamic Allocation	Disabled
Run Time	1 m 14 s 711 ms
Efficiency	76.62%

Figure 6: Executor usage with AutoExecutor as seen on diagnostic panel of an Azure Synapse Spark pool.

further provide visualizations of executor allocation and utilization as shown in Figure 6 for TPC-DS query 94. We can see that the allocated executors (blue line) increase automatically from the default value of 2 to 27 after we request the additional executors during query optimization. This also leads to better overall performance.

4.3 Scenario 3: tuning executor count for optimal query performance

Picking the "right" executor count is a challenge for Spark users. Therefore, users either keep the default executor count configuration of the cluster or supply a fixed value for all queries in the application. Unfortunately, such a static approach is sub-optimal for many Spark applications [21]. Therefore, in this scenario, we will demonstrate how AutoExecutor can automatically select the optimal executor count in the PCC before the performance flattens. This value could be set during query optimization as described above, or also during Spark application submission (e.g., by predicting the PCCs of all queries in the application upfront).

Figure 6 shows the executor usage for TPC-DS query 94, with executor count selected during query optimization by AutoExecutor. This results in a run time of 1m14s and executor efficiency of 76.62%. Alternatively, if AutoExecutor sets the executor count during application submission, then we get a run time of 1m7s and executor efficiency of 58.27%. In contrast, with dynamic allocation (configured with a range of 2–38 executors), we get a run time of 1m30s and executor efficiency of 62.14%. Finally, default and static allocation of 4 executors has the slowest run time of 3m15s and highest executor efficiency of 83.7%. During the demonstration, we will invite the audience to submit queries and show them the query performance and resource efficiency trade-offs involved.

4.4 Scenario 4: adapting to changing data and query characteristics

Production workloads are constantly evolving in terms of data and query characteristics. Given the feature-based model, AutoExecutor can adapt to such changes. We will demonstrate this scenario by encouraging the audience to enter modified queries and datasets and get the correspondingly adapted PCCs, without executing the queries. Our approach can be used to *forecast* future resources based on the expected data or query changes. The audience can select a trend for data size or for query plan size, and AutoExecutor will predict the growth in CPU core requirements over time.

4.5 Scenario 5: selecting cluster configurations

Spark's dynamic allocation feature grows the executor count requests exponentially and hence it suffers from asking too many executors and having other applications wait for resources. AutoExecutor avoids this by predicting the right size and letting other queries utilize the spare executors. Furthermore, given the PCCs from one or more applications, AutoExecutor can assist in choosing better VM configurations by combining the CPU core predictions from the PCCs of all queries of all applications on the cluster. Towards the end of the demonstration session, we will show the audience the recommended number of cores per node for the Spark SQL queries that they have played with in their session.

REFERENCES

[1] 2017. *Add hooks and extension points to Spark.* Retrieved July 26, 2021 from https://issues.apache.org/jira/browse/SPARK-18127
[2] 2019. *Koalas: Easy Transition from pandas to Apache Spark.* Retrieved July 26, 2021 from https://databricks.com/blog/2019/04/24/koalas-easy-transition-from-pandas-to-apache-spark.html
[3] 2020. *Apache Spark in Azure Synapse Analytics.* Retrieved July 26, 2021 from https://docs.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-overview
[4] 2020. *Qubole Sparklens tool for performance tuning Apache Spark.* Retrieved July 26, 2021 from https://github.com/qubole/sparklens v0.3.2.
[5] 2020. *What is Apache Spark in Azure HDInsight.* Retrieved July 26, 2021 from https://docs.microsoft.com/en-us/azure/hdinsight/spark/apache-spark-overview
[6] 2021. *Amazon Athena.* Retrieved July 26, 2021 from https://aws.amazon.com/athena
[7] 2021. *Azure Synapse Analytics.* Retrieved July 26, 2021 from https://azure.microsoft.com/en-us/services/synapse-analytics
[8] 2021. *BigQuery.* Retrieved July 26, 2021 from https://cloud.google.com/bigquery
[9] 2021. *Data Lake: Microsoft Azure.* Retrieved July 26, 2021 from https://azure.microsoft.com/en-us/solutions/data-lake
[10] 2021. *Dynamic Resource Allocation.* Retrieved July 26, 2021 from https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation
[11] 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.
[12] 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.
[13] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *25th IEEE International Conference on Data Engineering.* IEEE, 592–603.
[14] 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.
[15] Alekh Jindal, Shi Qiao, Rathijit Sen, and Hiren Patel. 2021. Microlearner: A fine-grained Learning Optimizer for Big Data Workloads at Microsoft. In *37th IEEE International Conference on Data Engineering.* 2423–2434.
[16] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. 2016. Morpheus: Towards automated SLOs for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation.* 117–134.
[17] 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. (2021). arXiv:2107.08594
[18] 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.
[19] 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).
[20] 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.
[21] 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.