



# APOENA: Towards a Cloud Dimensioning Approach for Executing SQL-like Workloads Using Machine Learning and Provenance

Raslan Ribeiro<sup>1</sup>, Rafaelli Coutinho<sup>2</sup><sup>a</sup> and Daniel de Oliveira<sup>1</sup><sup>b</sup>

<sup>1</sup>Institute of Computing, Universidade Federal Fluminense, Niterói, Brazil

<sup>2</sup>Federal Center for Technological Education Celso Suckow da Fonseca, Brazil

**Keywords:** Cloud Dimensioning, Query Execution Time, Machine Learning, Provenance Data, Big Data.

**Abstract:** Over the past decade, data production has accelerated at a fast pace, posing challenges in processing, querying, and analyzing huge volumes of data. Several platforms and frameworks have emerged to assist users in handling large-scale data processing through distributed and HPC environments, including clouds. Such platforms offer a plethora of cloud-based services for executing workloads efficiently in the cloud. Among these workloads are SQL-like queries, the focus of this paper. However, leveraging these platforms usually requires users to specify the type and number of virtual machines (VMs) to be deployed in the cloud. This task is not straightforward, even for expert users, as they must choose the VM type and number from several options available in a cloud provider's catalog. Although autoscaling mechanisms can be available, non-expert users may find it challenging to configure them. To assist non-expert users in dimensioning the cloud environment for executing SQL-like workloads in such platforms, *e.g.*, Databricks, this paper introduces a middleware named APOENA, which is designed to dimension the cloud for specific SQL-like workloads by collecting provenance data. These data are used to train Machine Learning (ML) models capable of predicting query performance for a particular combination of query characteristics and VM configuration.


## 1 INTRODUCTION


In recent years, many approaches have been proposed for processing and querying the so-called *Big Data*, and they aim to support the user in the decision-making process (Meredino et al., 2018). Some of these approaches are designed for local or cloud-based deployment and usage, such as Apache Spark<sup>1</sup> and Apache Hive<sup>2</sup>, while others are inherently cloud-based, like the services provided by major cloud providers. Despite representing a step forward, these approaches have limitations for some types of usage, *e.g.*, certain approaches lack support for complex queries or are unsuitable for transactional workloads.

Nevertheless, other solutions like Databricks (Zaharia, 2019), a data analytics platform, offers several advantages such as support for complex queries, transactional workloads, and autoscaling of workloads. Despite the appeal of these features, it is

worth noticing that Databricks is not provided free of charge, *i.e.*, the financial cost of its services can be substantial depending on the usage scenario. For instance, Databricks provides a query engine named *Photon*, optimized for accessing and querying data in a Data Lake (Nargesian et al., 2023). As of December 23rd, 2023, the financial cost<sup>3</sup> of using Photon (*i.e.*, DLT Advanced Compute Photon) on an m5dn.4xlarge AWS virtual machine (VM) (with 16vCPUs and 32GiB RAM) for just one hour per day over 30 days amounts to US\$102.10. This value can increase in a real-world production environment.

The problem is that most cloud providers offer an extensive variety of VM types, often more than a hundred. Each VM type has its specific characteristics, including the number of vCPUs, memory in GiBs, and financial cost in US dollars. Choosing the appropriate VM type based on the specific characteristics of a SQL-like query can pose a challenge. For instance, consider a SQL-like query containing the predicate `PERSON_ID = 44311`, which restricts the

<sup>a</sup> <https://orcid.org/0000-0002-1735-1718>

<sup>b</sup> <https://orcid.org/0000-0001-9346-7651>

<sup>1</sup><https://spark.apache.org/>

<sup>2</sup><https://hive.apache.org/>

<sup>3</sup><https://www.databricks.com/br/product/pricing/product-pricing/instance-types>

possible values of the `PERSON_ID` attribute to a single value. This condition establishes a specific *selectivity factor*<sup>4</sup>. Introducing a different predicate, such as `PERSON_ID IN (44311, 44556, 33245)`, can alter the selectivity factor and consequently affect the query execution time. Additionally, the number of tuples in one or more tables probably impacts the query execution time. Therefore, determining the suitable VM type for executing a specific query (or a set of queries) is far from trivial. Poor execution of this task may result in financial costs due to over-dimensioning or performance issues stemming from under-dimensioning of the cloud environment. Although Databricks offers autoscaling mechanisms, it can be a tricky task to be accomplished by non-experts.

This paper introduces `APOENA` (word from the Tupi language that means “the one who sees further”), a middleware designed to dimension the cloud to specific SQL-like workloads in platforms like Databricks. `APOENA` collects a set of metadata, including provenance data (Herschell et al., 2017) — *i.e.*, the historical record of previously executed queries. The goal is to use this information to train Machine Learning (ML) models capable of predicting the execution time of an SQL-like query in a specific configuration of the virtual cluster to help the user configure the platform. Through learning from provenance, `APOENA` aims to prevent both over-dimensioning and under-dimensioning of the cloud environment. It is worth noting that `APOENA` is complementary to the Databricks platform and is not designed to replace existing mechanisms.

This paper is structured into four sections, besides this introduction. Section 2 discusses the background and delves into related work. Section 3 introduces `APOENA`, while Section 4 evaluates the proposed approach. Lastly, Section 5 concludes the paper, summarizing key findings and highlighting future work.

## 2 BACKGROUND

### 2.1 Databricks Platform

The Databricks platform is designed for processing large-scale data using big data frameworks such as Apache Spark and storing these data in a Delta Lake (Armbrust et al., 2020) as depicted in Figure 1. The software stack available for this data processing is called *Databricks Runtime*. The data manipulation

can be performed using Apache Spark, an engine designed for processing data in both single nodes and distributed clusters, but Databricks runtime also incorporates other components, *e.g.*, Photon (Behm et al., 2022). Photon serves as a specialized query engine designed for Lakehouse environments (Zaharia et al., 2021). Photon leverages the advantages of Delta Lake (Armbrust et al., 2020), which stores files in delta type. Delta files, built on top of parquet files, offer features such as data versioning and support for upsert operations (*i.e.*, update and insert). The Unity Catalog in this software ecosystem also plays a pivotal role by centralizing data control, simplifying access, and easing auditing processes.

Databricks offers Delta Live Tables (DLT), enabling pipeline construction by defining data transformations using SQL or Python. DLT automates lineage generation, linking tables to their data dependencies, and ensuring each table is generated after its predecessors. With all workloads planned for cloud processing (*e.g.*, AWS or Azure), sizing the virtual cluster appropriately is crucial for performance and cost. Databricks provides autoscaling mechanisms for dynamic cluster sizing, though it relies on reactive algorithms based on performance metrics. Determining resource needs can be complex, especially for non-expert users unfamiliar with autoscaling configurations. This paper’s approach aims to guide non-expert users in determining an effective initial cluster size to avoid under or over-sizing or minimize adjustments from default autoscaling configurations, rather than achieving globally optimal sizing or catering to expert optimization efforts.

### 2.2 Provenance Data

Provenance is commonly defined as “the history of data” (Herschell et al., 2017). This term denotes the metadata that explains the process of generating a specific piece of data. Its purpose is to systematically record, in a structured and queryable format, the data derivation path within a particular context. Initially used for assessing quality and fostering reproducibility in scientific experiments, its application extends beyond its original purpose. In the context of this paper, it serves as a rich source of information, encompassing consumed parameter values and execution times of SQL-like queries submitted by users. In this regard, provenance data can be employed for dimensioning the cloud environment, which is the primary focus of this paper.

While various methods exist for representing and storing provenance data, the W3C recommendation, known as PROV (Groth and Moreau, 2013), defines

<sup>4</sup>The ratio of qualifying tuples to the total number of tuples in the query.

a data model for this purpose. PROV conceptualizes provenance with *Entities*, *Agents*, *Activities*, and various relationship types. An *Entity* represents a tangible object or concept, such as a SQL-like query or a database table with parameter values. *Activities* are actions within Databricks affecting entities, like query execution, with associated times and errors. Lastly, an *Agent* is a user executing activities. Despite being domain-agnostic, PROV can extend to various fields, including those addressed in this paper.

The specification of a query can be viewed as *Prospective Provenance (p-prov)*, a form of provenance data that logs the steps carried out during data processing. Another category of provenance is *Retrospective Provenance (r-prov)*, which captures details related to the execution process. This includes information such as when a query is submitted and executed, the duration of its execution, the parameters used, any errors encountered, *etc.*

### 2.3 Related Work

Several papers have previously proposed methods for dimensioning the virtual cluster or predicting SQL query execution time in specific environments using different big data frameworks (de Oliveira et al., 2021; Mustafa et al., 2018; Burdakov et al., 2020). Singhal and Nambiar (2016) introduce an analytical model that dynamically adjusts the query execution plan according to some performance metrics. This is achieved by constructing individual ML models to estimate the database and operating system's cache, as well as the execution time of each SQL operator in the query plan. Such individual models are further combined into one. Mustafa et al. (2018) propose an ML model capable of predicting the execution time of Spark jobs. The evaluation of the proposed model is based on metrics such as R-Squared, Adjusted R-Squared and Mean Squared Error. The authors consider the dataflow created by Spark to be composed of multiple tasks and each stage is based on data partitioning. After each task, various features, such as input and output data sizes, are used to train the ML model. Following each stage (composed of multiple tasks), the number of transformations is used as input for the prediction model. Although this approach represents a step forward, it is focused on Spark jobs, as it consumes features specific to Spark, such as data partitions in RDD and dataframes.

Burdakov et al. (2020) propose a cost model for query time estimation for use in Database as a Service (DaaS) platforms. The approach proposes employing a Bloom filter and duplicating small tables across multiple nodes to reduce query execution time.

The cost model assumes as a promise that there is a uniform distribution of attribute values, independence of attributed values, and that keys from a small domain can be found in large domains. Ahmed et al. (2022) compare various analytical and ML models for the estimation of runtime in big data jobs. The authors conclude that the choice of the most suitable machine learning or analytical approach depends on the specific type of job and the environment in which it is submitted to. Öztürk (2023) introduces a multi-objective optimization method for tuning Spark-based jobs. The author demonstrates that incorporating data compression and memory usage features enhances the effectiveness of multi-objective optimization methods specifically designed for Spark. Likewise, de Oliveira et al. (2021) propose the usage of Decision Trees classifiers for predicting the execution time of Spark jobs, aiming to assist in parameter tuning. Also, Filho et al. (2021) propose an approach for Hadoop tuning that can provide good performance for Hive queries.

While the aforementioned approaches represent a step forward, they primarily focus on specific DBMSs or big data frameworks. To the best of the authors' knowledge, there is no specific approach tailored to recommending the suitable amount of resources for SQL-like queries in platforms like Databricks.

## 3 THE APOENA MIDDLEWARE

This section introduces APOENA, a middleware integrated into the Databricks runtime service, specifically designed for cloud environment dimensioning for SQL-like query workloads. APOENA comprises five key components, as presented in Figure 1: (i) Crawler, (ii) Workload Generator, (iii) Statistics Database, (iv) Model Trainer, and (v) Dimensioner. Subsequently, we delve into the specifics of each of these components.

To execute, APOENA requires a series of queries to have been previously submitted and executed. These queries can range from simple ones like "SELECT \* FROM t1" to complex queries involving multiple tables, aggregation functions, DLT, and the usage of Photon. The *Crawler* is responsible for collecting all information related to the previously executed queries. It accesses the logs in Databricks platform to extract the provenance associated with the queries, *e.g.*, execution time, Photon usage, the number of input tuples, worker type, the number of workers *etc.*

It is noteworthy that the *Crawler* also collects metadata related to the ETL routines, the executed queries and their respective executors, *i.e.*, provenance data. In many existing ETL routines, data are

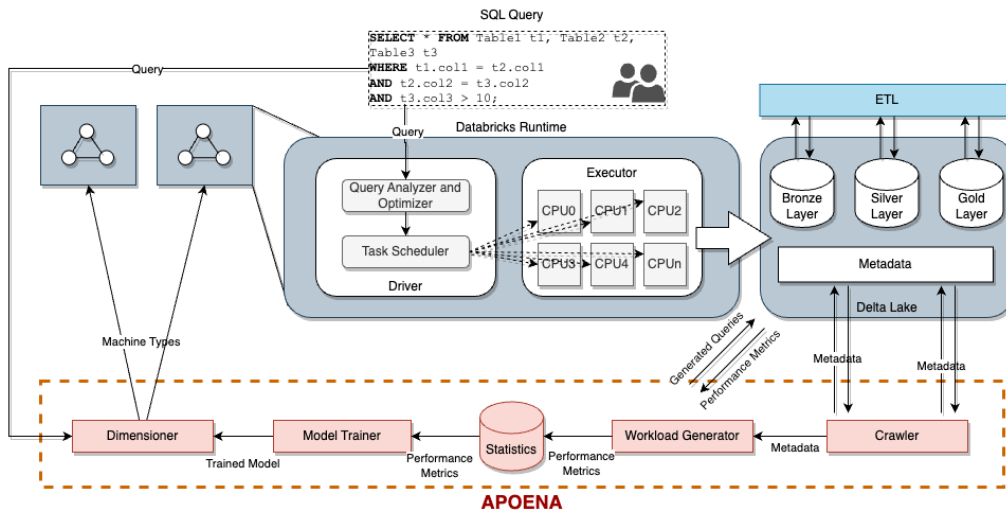


Figure 1: The Architecture of APOENA middleware. Gray components are the ones already provided by Databricks runtime Service and Delta Lake while red components are part of APOENA.

organized into three layers: (i) Bronze Layer, (ii) Silver Layer, and (iii) Gold Layer. The data in the Bronze Layer does not differ significantly from the raw data in the Lakehouse; the distinction lies only in the representation, wherein data from different formats are loaded into a table. Data in the Silver Layer is already transformed and cleaned. Additionally, the data can be enriched with more useful information, such as replacing country name abbreviations with the full name or obtaining the complete address of a place based on its postal code. In the Gold Layer, data is stored in data marts for decision-making. The layer associated with each query is captured by the *Crawler*.

Following this, the *Crawler* sends the captured metadata to the *Workload Generator*. This component is employed to generate representative queries in case the number of past executions is small (e.g., less than 500 past executions). Since we assume that most of the executed queries will access data stored in the Gold Layer, we can generate multiple representative queries to increase the number of past executions for APOENA. In this sense, we have adapted the algorithm for workload generation previously proposed by Ortiz et al. (2015), outlined in Algorithm 1. The input consists of a database following a star or snowflake schema (i.e., the database in the Gold Layer), denoted as  $D = \{f \cup \{dim_1, dim_2, \dots, dim_k\}\}$ , comprising one fact table ( $f$ ) and multiple dimension tables. It is noteworthy that we assume the fact table has multiple foreign keys to dimension tables. Each table  $t_i \in D$  is composed of a set of attributes  $Att(t_i)$ . The algorithm produces a set of possible queries  $Q$  to be executed and the associated performance metrics  $M$  collected (the generated queries may not represent a real-world

analysis to be performed by users, but it can be used as an example for training the ML models following). Each  $q_i \in Q$  is associated with a tuple  $(T^q, A^q, e^q)$  where  $T^q$  is the set of tables in the FROM clause (i.e., the fact table and one or more dimension tables),  $A^q$  is the set of projected attributes in the SELECT clause, and  $e^q$  is the selectivity factor. As previously defined by Ortiz et al. (2015), the algorithm keeps a list  $L$  with all representative sets of tables  $T^q$ . Initially, every single table  $t_i \in D$  is inserted into  $L$  (lines 4-6), then the algorithm generates the joins (line 8). When all representative queries are produced, the algorithm starts executing each  $q_z \in Q$  using different cluster configurations, i.e., different types of VMs (lines 26-28).

The collected provenance data for each query execution includes: (i) query ID, (ii) number of input tuples, (iii) number of output tuples, (iv) number of vCPUs in the worker, (v) memory (in GB) of the worker, (vi) selectivity factor, (vii) number of workers, (viii) number of attributes, (ix) if the query was accelerated using Photon, (x) if the query has an inner join, (xi) if the query has a left join, (xii) the layer (i.e., bronze, silver, or gold), (xiii) if the query has a GROUP BY clause, and (xiv) the execution time. Once all provenance data regarding the executed queries is available, the *Model Trainer* can be invoked. The *Model Trainer* uses the received provenance data to train ML models for use in the cloud dimensioning task. In its current version, APOENA uses Linear Regression (LiR), Logistic Regression (LoR), Decision Tree Classifier (DTC), and Random Forest Regression (RFR), all available in scikit-learn<sup>5</sup>. LiR aims to fit the results into a linear

<sup>5</sup><https://scikit-learn.org/stable/>

Algorithm 1: Workload Generation.

---

```

1   $Q \leftarrow \{\}$ ;
2   $L \leftarrow \{\}$ ;
3   $M \leftarrow \{\}$ ;
4  foreach  $t_i \in D$  do
5       $T^q \leftarrow \{t_i\}$ ;
6       $L \leftarrow L \cup T^q$ ;
7      if  $isFact(t_i)$  then
8           $SortDesc(\{dim_1, \dots, dim_k\}), 1 \leq i \leq k$ ;
9          foreach  $j, 1 \leq j \leq k$  do
10              $D^j \leftarrow$  first  $j$  tables from  $D$ ;
11              $T^q \leftarrow T^q \cup D^j$ ;
12              $L \leftarrow L \cup T^q$ ;
13         end
14     end
15 end
16 foreach  $T^q \in L$  do
17      $SortDesc(Att(T^q), Att_j(T^q) \leq j \leq Att(T^q))$ ;
18     foreach  $k, 1 \leq k \leq Att(T^q)$  do
19          $A^q \leftarrow$  first  $k$  attributes from  $Att(T^q)$ ;
20         foreach  $e^q \in E_{T^q}$  do
21              $Q \leftarrow Q \cup (T^q, A^q, e^q)$ ;
22         end
23     end
24 end
25 foreach  $q_z \in Q$  do
26     foreach  $vm_c \in VM$  do
27          $P^{q_z} \leftarrow Perf(q_z, vm_c)$ ;
28          $M \leftarrow M \cup P^{q_z}$ ;
29     end
30 end

```

---

equation by minimizing the sum of the squared differences between predicted and actual values (Russell and Norvig, 2020). Unlike LiR, LoR works with one or more independent variables based on finding a relationship between them. DTC is based on learning the final classification according to its features, developing a prediction model with rules based on previous results (Russell and Norvig, 2020). Finally, RFR comprises sample groups of decision tree classifiers that estimate an average classifier according to each sample (Russell and Norvig, 2020).

Once the ML models are trained, the *Dimensioner* can be invoked. The idea behind the *Dimensioner* is that every time a new query is submitted, APOENA identifies the characteristics of this query and combines them with all possible virtual cluster configurations (*i.e.*, the types and number of VMs). Each combination of query and environment setup is then submitted to each trained ML model for inference. The ML model responds with the expected execution time of the query for that specific virtual cluster configuration. The *Dimensioner* then orders and presents the top K configurations that will execute the query faster and/or with less financial cost, depending on the

goal defined by the user. The user can then dimension the environment by choosing the most suitable virtual cluster configuration. It is worth mentioning that APOENA does not replace any existing mechanism, such as autoscaling. Instead, the idea is to avoid unnecessary deployments and undeployments of VMs in the process and to assist especially non-expert users.

## 4 EXPERIMENTAL EVALUATION

This section evaluates APOENA in a real production scenario. First, we present the chosen evaluation metrics. Then, we discuss how the computational environment was configured. Following that, we present the data used in the experiments, and finally, we discuss the achieved results.

### 4.1 Metrics

Metrics must be employed to evaluate the performance of trained ML models to predict the query execution time for a specific query and virtual cluster configuration. They include True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN). A TP is a result in which the model correctly predicts the correct class. Similarly, a TN is a result in which the model correctly predicts the negative class. An FP is a result in which the model incorrectly predicts the positive class. A FN is a result in which the model incorrectly predicts the negative class. Based on that, we can define the *Accuracy* as the number of correct predictions (TP+TN) divided by the total of predictions (TP+TN+FP+FN). We can also use the *F1\_score*, which represents the harmonic mean between precision ( $Precision = \frac{TP}{TP+FP}$ ) and recall ( $Recall = \frac{TP}{TP+FN}$ ), as defined by  $F1\_score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$ . However, the *F1\_score* can involve a micro, macro, or weighted average. According to Plaue, the micro average is a global metric that considers the total of true positives, false negatives, and false positives, as  $F1\_micro = \frac{TP}{TP + \frac{FP+FN}{2}}$ .

On the other hand, the macro average is not a global metric. It calculates the *F1\_score* for each class  $i$  and determines their unweighted mean, where  $n$  is the number of classes as defined by  $F1\_macro = \frac{\sum_{i=1}^n F1\_score_i}{n}$ . Thus, it ignores unbalanced data. Finally, the weighted average considers unbalanced data, calculating a *F1\_score* for each class  $i$  separately and its respective weight  $w_i$  (Kundu, 2022), such that:  $w_i = \frac{Quantity\ of\ samples\ in\ class\ i}{Total\ quantity\ of\ samples}$ , thus  $F1\_weighted = \sum_{i=1}^n w_i \times F1\_score_i$ .

## 4.2 Environment Setup

When configuring the environment, the user has to choose a plethora of parameters in Databricks platform, such as the Databricks runtime version, Photon acceleration, number of workers and VM types to be deployed. Our experiments are based on Databricks version 12.2 LTS (using Scala 2.12 and Spark 3.3.2). The VM types used are a subset of the ones provided by AWS<sup>6</sup>. We considered only the information about the number of vCPUs, RAM (in GB) and storage size (in GB) of each VM type. Among many options, the chosen ones were selected based on the best cost-benefit, as presented in Table 1.

Table 1: Chosen AWS VM Types.

VM type	vCPU	RAM (GB)	Storage (GB)
r5d.xlarge	4	32	1 x 150
r5d.2xlarge	8	64	1 x 300
i3.xlarge	32	32	1 x 950
m5d.4xlarge	64	64	2 x 300

## 4.3 Experiment Setup

All data used in the experiments were collected from real-world SQL-like queries submitted to Databricks in a large-scale company. The queries are associated with LinkedIn profile data and Internal Revenue Service data. Such data sources provide rich information that helps data analysts determine the profile of companies and individuals based on the provided content. The used database includes the following attributes: (i) employer identification number, (ii) share capital, (iii) address, (iv) number of employees, (v) business segment, (vi) revenue, (viii) email, (ix) telephone number, *etc.* It is worth mentioning that the used data did not have sensitive information.

A set of provenance data associated with each query execution was collected so it is possible to conduct the evaluations and can be divided into four classes: (i) query structure (*i.e.*, *p-prov*), (ii) consumed/produced data information (*i.e.*, *r-prov*), (iii) Databricks setup and (iv) computational environment (*i.e.*, *r-prov*). Metadata regarding the query structure was collected by analyzing each query individually to retrieve information about the usage of specific SQL statements, such as constraints, CTE, CASE WHEN, joins (*e.g.*, INNER JOIN, LEFT JOIN, and RIGHT JOIN), GROUP BY, subqueries, EXPLODE and the selectivity factor. The latter is calculated by determining the proportion of the filtered data analyzed against the total number of tuples in the tables.

Metadata about the tables was collected, including

<sup>6</sup><https://aws.amazon.com/ec2/instance-types/>

the number of input rows, input and output attributes (*i.e.*, projection). Concerning the computational environment and the number of vCPUs information in each VM, the RAM amount (in GBs) in each VM and the storage capacity (in GBs) in each VM were gathered. Finally, concerning the Databricks setup, information on the number of workers, the type of each worker and whether the query used Photon for acceleration was collected.

We also specify the ML algorithms that will be employed to train the ML models by APOENA. Since we are using scikit-learn as the ML framework, we have chosen the following algorithms: LiR, LoR, DTC and RFR. Each trained ML model is submitted to the 10-fold Cross-Validation method for evaluation of the *F1\_weighted* metric.

## 4.4 Results

Since in APOENA, we aim to predict the execution time of a SQL-like query based on characteristics of the query and the computational environment used, the *query execution time* is our target class. As the execution time of queries is rarely the same, it is preferable to discretize this class into a series of execution time intervals. Our first task is to choose a suitable number of intervals for query execution time. Table 2 presents eight possible discretizations of the query execution time that can be used by APOENA.

The dataset, containing query provenance and Databricks environment features, becomes unbalanced depending on chosen execution time intervals. Real-world data from a production system shows the majority of queries execute in under 600 seconds. Thus, fewer intervals lead to a more unbalanced dataset, with most queries falling into the first interval. Hence, we evaluate scenarios with six, seven, and eight execution time intervals. Figure 2(a) shows the percentage of examples per time interval for a scenario with eight intervals.

After defining the number of execution time intervals, we evaluated the importance of features in the dataset. Selecting the most relevant features based on their impact on the predicted result is essential for APOENA. Measures based on the impurity reduction of splits in DTC are common since they are simple and fast to compute. Thus, we have chosen to use Gini importance (Nembrini et al., 2018). Figure 2(b) presents the importance of each feature.

Differently from what we expected, some features presented low importance in the dataset, such as the use of LEFT JOIN and INNER JOIN clauses in the queries. This could be due to the fact that only one query includes a LEFT JOIN, which reduces its global

Table 2: Intervals for Query Execution Time (in seconds).

Number of Intervals	Intervals (in seconds)
2	0-1800; > 1800
3	0-300; 300-3600; > 3600
4	0-300; 300-1800; 1800-7200; > 7200
5	0-300; 300-1800; 1800-7200; 7200-14400; > 14400
6	0-300; 300-1800; 1800-3600; 3600-7200; 7200-14400; > 14400
7	0-300; 300-600; 600-1800; 1800-3600; 3600-7200; 7200-14400; > 14400
8	0-30; 30-300; 300-600; 600-1800; 1800-3600; 3600-7200; 7200-14400; > 14400

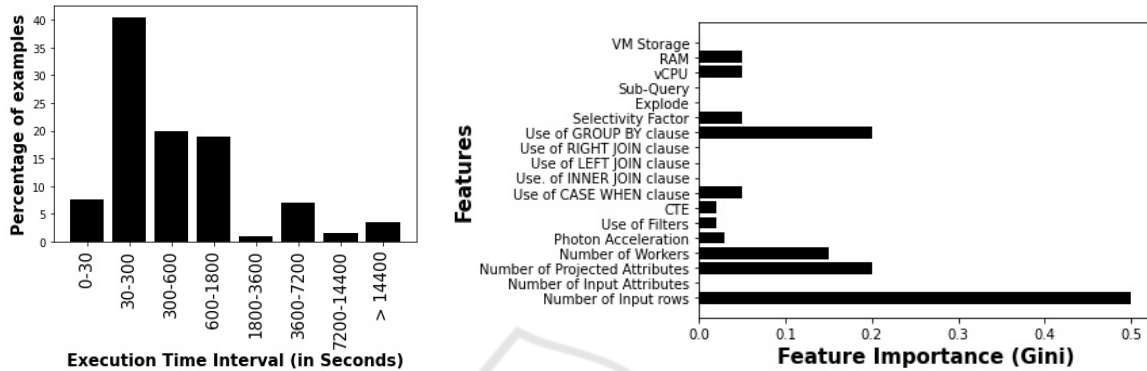


Figure 2: (a) Percentage of examples per execution time interval (8 intervals). (b) Feature Importance.

importance despite adding to the complexity of that query. However, as expected, the number of tuples processed by the query and characteristics of the environment, e.g., the number of vCPUs, and the number of workers have an impact on the result. All features with importance greater than zero were considered in the following analyses.

We have executed each ML algorithm to train the models for 6, 7, and 8 execution time intervals, evaluating the accuracy and the F1\_weighted metrics of each one. By analyzing Table 3, one can observe that the DTC is the classifier that showed the highest accuracy and F1\_weighted for all three execution time intervals. We have set the minimum accepted values for accuracy and F1\_weighted at 0.9 for both metrics. Therefore, the DTC was the chosen ML model for use in this dataset by APOENA. It is worth noting that a different classifier may be chosen if the query patterns change drastically over time. Thus, the ML models have to be retrained periodically in APOENA. The automatic retraining feature is still under development.

Table 3: Performance metrics according to time intervals.

Intervals	Metric	LiR	LoR	RFR	DTC
6	Accuracy	0.70	0.36	0.88	0.94
	F1_weighted	0.77	0.19	0.91	0.94
7	Accuracy	0.46	0.22	0.80	0.84
	F1_weighted	0.57	0.08	0.81	0.84
8	Accuracy	0.42	0.22	0.64	0.76
	F1_weighted	0.48	0.08	0.67	0.76

We also assessed the impact of correct and incorrect recommendations made by APOENA for two representative queries (transformations from bronze to the silver layer). The first one accessed 332,263,103 tuples, producing 41,041,580 tuples as output. This query is executed in a virtual cluster composed of two VMs r5d.2xlarge for 1,219 seconds. APOENA correctly classified it into the 300-1800 execution time interval and the query was executed within the predicted time, costing US\$ 4.87. On the contrary, the second one accessed 1,327,822,307 tuples, producing 415,478,474.00 tuples as output. The optimal configuration for a balance of performance and cost was a virtual cluster composed of 10 VMs m5d.4xlarge, running for 1,776.00 seconds and costing approximately US\$ 35.52. However, APOENA classified the execution time interval for this query as 14,400 seconds, which may make the user choose a different virtual cluster configuration. We only considered the VM types presented in Table 1. If more powerful VMs are available for APOENA, the user might choose an over-dimensioned configuration incurring higher financial costs. Finally, we evaluated the overhead imposed by APOENA to dimension the virtual cluster. On average, APOENA needed 100 seconds to train ML models and less than a second for inference, which can be considered an acceptable overhead.

## 5 CONCLUSIONS

Several big data processing platforms have emerged, with Databricks standing out as one of the most prominent options. It offers a range of cloud-based services for executing complex queries and transactional workloads. However, it requires the dimensioning of the cloud environment, demanding users to specify the types and number of VMs for deployment, a task that can be far from trivial. Identifying the characteristics of a SQL-like workload and estimating the appropriate VM type from a selection of more than 100 options poses a complex challenge. While mechanisms like autoscaling exist in Databricks, they are costly and may not be straightforward to configure for non-expert users. Improperly dimensioning the cloud environment, either through over or under-dimensioning, can impact both workload performance and financial costs.

This paper proposes a middleware named APOENA, designed to dimension the cloud environment for specific workloads, *i.e.*, SQL-like queries. APOENA collects provenance data and employs this historical data to train ML models capable of predicting query performance for a particular combination of a query and virtual cluster configuration. This configuration includes the type and number of VMs involved in the execution. Although APOENA focused on Databricks in this paper, it could be extended to work with other big data frameworks such as Apache Spark.

Experiments with real-world workloads demonstrated that APOENA classified query execution times with over 90% accuracy and F1-weighted metrics. Future work involves implementing a retraining mechanism for APOENA, as the current version does not perform retraining automatically. Additionally, we plan to evaluate APOENA using a broader range of real-world workloads.

## ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This paper was also partially financed by CNPq (grant n<sup>o</sup> 311898/2021-1) and FAPERJ (grant n<sup>o</sup> E-26/202.806/2019).

## REFERENCES

Ahmed, N. et al. (2022). Runtime prediction of big data jobs: performance comparison of machine learning al-

- gorithms and analytical models. *J. Big Data*, 9(1):67.
- Armbrust, M. et al. (2020). Delta lake: High-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.*, 13(12):3411–3424.
- Behm, A. et al. (2022). Photon: A fast query engine for lakehouse systems. In *SIGMOD'22*, pages 2326–2339. ACM.
- Burdakov, A. et al. (2020). Predicting sql query execution time with a cost model for spark platform. In *IoTBDs'20*, pages 279–287. INSTICC, SciTePress.
- de Oliveira, D. E. M. et al. (2021). Towards optimizing the execution of spark scientific workflows using machine learning-based parameter tuning. *Concurr. Comput. Pract. Exp.*, 33(5).
- Filho, E. R. L., de Almeida, E. C., Scherzinger, S., and Herodotou, H. (2021). Investigating automatic parameter tuning for sql-on-hadoop systems. *Big Data Res.*, 25:100204.
- Groth, P. and Moreau, L. (2013). W3C PROV - An Overview of the PROV Family of Documents. Available at <https://www.w3.org/TR/prov-overview/>.
- Herschell, M., Diestelkämper, R., and Lahmar, H. B. (2017). A survey on provenance: What for? what form? what from? *The VLDB Journal*.
- Kundu, R. (2022). F1 score in machine learning: Intro & calculation.
- Meredino, A. et al. (2018). Big data, big decisions: The impact of big data on board level decision-making. *Journal of Business Research*.
- Mustafa, S., Elghandour, I., and Ismail, M. A. (2018). A machine learning approach for predicting execution time of spark jobs. *Alexandria Engineering Journal*, 57(4):3767–3778.
- Nargesian, F., Pu, K. Q., Bashardoost, B. G., Zhu, E., and Miller, R. J. (2023). Data lake organization. *IEEE Trans. Knowl. Data Eng.*, 35(1):237–250.
- Nembrini, S., König, I. R., and Wright, M. N. (2018). The revival of the gini importance? *Bioinform.*, 34(21):3711–3718.
- Ortiz, J., de Almeida, V. T., and Balazinska, M. (2015). Changing the face of database cloud services with personalized service level agreements. In *CIDR'15*.
- Öztürk, M. M. (2023). Tuning parameters of apache spark with gauss-pareto-based multi-objective optimization. *Knowledge and Information Systems*.
- Plaue, M. (2020). *Data Science - An Introduction to Statistics and Machine Learning*. Springer.
- Russell, S. J. and Norvig, P. (2020). *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson.
- Singhal, R. and Nambiar, M. (2016). Predicting sql query execution time for large data volume. In *IDEAD'16*, page 378–385, New York, NY, USA. Association for Computing Machinery.
- Zaharia, M. (2019). Lessons from large-scale software as a service at databricks. In *SoCC'19*, page 101.
- Zaharia, M. et al. (2021). Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *CIDR'21*. [www.cidrdb.org](http://www.cidrdb.org).