Expand your Training Limits! Generating Training Data for ML-based Data Management

Francesco Ventura*
francesco.ventura@polito.it
Politecnico di Torino

Zoi Kaoudi
zoi.kaoudi@tu-berlin.de
TU Berlin & DFKI GmbH

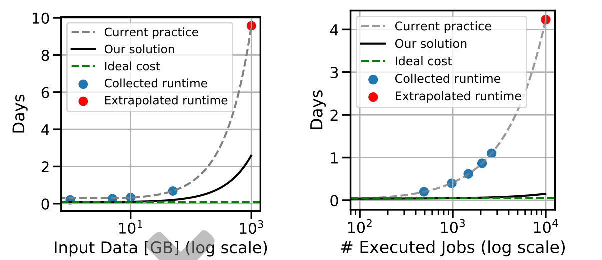
Jorge-Arnulfo Quiané-Ruiz
jorge.quiane@tu-berlin.de
TU Berlin & DFKI GmbH

Volker Markl
volker.markl@tu-berlin.de
TU Berlin & DFKI GmbH

ABSTRACT

Machine Learning (ML) is quickly becoming a prominent method in many data management components, including query optimizers which have recently shown very promising results. However, the low availability of training data (i.e., large query workloads with execution time or output cardinality as labels) widely limits further advancement in research and compromises the technology transfer from research to industry. Collecting a labeled query workload has a very high cost in terms of time and money due to the development and execution of thousands of realistic queries/jobs.

In this work, we face the problem of generating training data for data management components tailored to users' needs. We present DataFarm, an innovative framework for efficiently generating and labeling large query workloads. We follow a data-driven white-box approach to learn from pre-existing small workload patterns, input data, and computational resources. Our framework allows users to produce a large heterogeneous set of realistic jobs with their labels, which can be used by any ML-based data management component. We show that our framework outperforms the current state-of-the-art both in query generation and label estimation using synthetic and real datasets. It has up to 9× better labeling performance, in terms of R^2 score. More importantly, it allows users to reduce the cost of getting labeled query workloads by 54× (and up to an estimated factor of 104×) compared to standard approaches.



(a) 500 jobs, varying input size. **(b) 1GB input data, varying #jobs.**

Figure 1: Runtimes for manually collecting query labels.

1 INTRODUCTION

Machine Learning (ML) is rapidly acquiring a prominent role in data management, being at the core of several innovative techniques [10, 18, 19, 24–26], including different aspects of query optimization. For instance, researchers have used ML for cardinality estimation [23, 24, 46], join ordering enumeration [25, 31], cost model learning [2, 30], execution time estimation [3, 21, 33, 55], and for learning the entire query optimizer itself [30].

However, a well-known requirement for most ML-based solutions is the acquisition of valuable data to train the models on. The effectiveness of such models depends on (i) the quantity of training data, (ii) the quality of training data, i.e., the quality of the features that can be extracted, and (iii) the availability of valuable ground-truth labels to learn from. These requirements quickly become a road blocker in the context of query optimization: Collecting a large number of real queries with labels (e.g., jobs' execution time or cardinality values) is a tedious (if not impossible) task. It requires developing and executing thousands of heterogeneous queries (or jobs)[1] to generate training data composed of queries with labels.

For example, collecting execution times or cardinality values for 10,000 jobs with 1TB of data would require more than 6 months. Figure 1a shows that collecting labels for only 500 OLAP jobs with input data of about 1TB would take almost 10 days. Also, Figure 1b shows that, even with just 1GB of input data, running 10,000 jobs can easily go over four days.[2] Ideally, once the training data reaches good performance, its size does not have to increase over time if concept drift is not present, which is not true in reality [9, 16, 20, 22]. Therefore, the current practice of running an entire query workload to get its labels is not only expensive but also more than two orders of magnitude slower than the ideal. Furthermore, one has to collect again the labels whenever the distribution of the input data and workload changes [16, 19], making this approach impractical.

Surprisingly, the database community has made little progress in tackling the problem of generating labeled workloads [19, 21]. Most works still assume the availability of training data exploiting both private and public, often synthetic, datasets to develop and test their solutions [24, 25, 30, 53]. In contrast to other domains, where advanced data augmentation techniques play an important role [39, 42, 45], just a few preliminary attempts have been made in data management [21]. We still rely on task-specific benchmarking

*Work done while interning at TU Berlin.

[1] We henceforth use the two terms interchangeably.
[2] We provide the experimental setup in Section 6.1.

workload generators, such as TPC-H [37] or TPC-DS [36], which produce homogeneous workloads with low variance from few fixed patterns. However, none of these benchmarks provide any kind of labeling and thus they require executing a huge amount of queries.

We make a step forward towards the reliability of ML-based query optimization, where ML is used in the place of cost models or cardinality estimation processes, via a training data generation process. This can be achieved with a data-driven white-box query workload augmentation which includes label estimation for each query: (i) A data-driven approach can tailor the newly generated workload to users' needs by considering their workloads, input data, and computational resources; (ii) Estimating labels, i.e., execution times or cardinality values, avoids the onerous task of executing a workload of thousands of jobs, and; (iii) A white-box strategy allows users to understand and debug every step of the process.

However, developing an efficient and reliable training data generation framework is challenging for many reasons. First, how can we generate jobs that are representative of an existing small workload? Second, how can we take into account the real distribution of the data while generating a new labeled query workload? Third, how can we efficiently produce reliable labels taking into account the performance of computing resources? Fourth, which is the smallest set of representative queries that should be actually executed to learn information that provides reliable labels?

We propose DataFarm, a novel framework that enables a data-driven and white-box training data generation required for ML-based query optimization. In detail, we (i) analyze the execution patterns of an existing small query workload, (ii) analyze the distribution of input data and fitting the characteristics of computational resources, and (iii) significantly reduce the labeling generation time. In summary, after giving an overview of our framework (Section 2), we present the following major contributions:

(1) We propose a data-driven augmentation process of pre-existing query workloads. This process learns the real execution patterns as Markov Chains [15] and generates new heterogeneous abstract plans exploiting real operators' distributions. (Section 3)
(2) We present a job instantiation algorithm that allows us to create an augmented set of realistic jobs by instantiating different variants for each previously generated abstract plan. Also, it exploits the user's input data to tailor the newly generated workload to real use cases increasing its reliability. (Section 4)
(3) We introduce an efficient labeling forecasting process based on an active learning approach. It characterizes jobs at the operator-level with interpretable features, actively improving forecasting performance by executing the smallest number possible of jobs. It also predicts the uncertainty for each forecasted label, which allows us to optimize the accuracy of the process. (Section 5)
(4) We extensively evaluate DataFarm and demonstrate its quality and high efficiency compared to the current state-of-the-art. The results show the superiority of our framework: It has up to 9× better prediction performance and saves up to 54× of time by building a reliable dataset that is tailored to the user's need. (Section 6)

2 OVERVIEW

The goal of DataFarm is to produce, in a reasonable time, a large amount of training data (i.e., query workload with labels, typically runtime) required for learned data management components. Figure 2 illustrates the process, composed by: (i) the *Abstract Plan Generator*, (ii) the *Synthetic Job Instatiator*, and (iii) the *Label Forecaster*. We use throughout the paper the following running example. The example input data are three tables, `A.Products(id, price, description)`, `B.Shopping_Cart(product_id, customer_id, quantity)`, `C.Customers(id, age, name)`, referenced from now on as Table A, B, and C, and the fields of each table are enumerated with the corresponding lowercase letters and position id for brevity (e.g., `Products.id` is `A.a1`). We consider a query (a dataflow job) that filters the products with a price higher than 5, then joins tables A and B to obtain the products currently in the users' shopping cart, and aggregates the results to obtain the total price considering the ordered quantities. The black-edged nodes in the Directed Acyclic Graph (DAG) in Figure 3 show the execution plan of this job.

The *Abstract Plan Generator* (Section 3) takes as input the pre-existing (presumably small) real workload, learns the relations between the operators, and generates a number of new heterogeneous *abstract plans* 1. An abstract plan is a DAG of platform-agnostic operators that describes an execution plan without the actual operator implementation. In our example, a generated abstract plan follows a similar structure to the original plan while it may include new operators (operators without a black-edge in Figure 3). The user can specify settings, such as the maximum number of plans to generate and the maximum number of join operators in each plan.

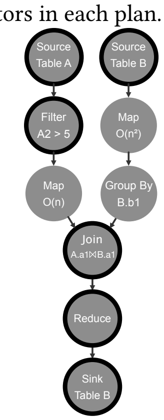


Figure 3: Example DAG.

Then, the *Synthetic Job Instantiator* (Section 4) takes as input the generated abstract plans and for each of them, it creates different *job instances* 2. It combines input parameters extracted from the input data metadata, such as the distribution of filterable values and input data entity relation schema. The content of the black-edged nodes in Figure 3 describes a possible instantiation. The output is an augmented set of realistic jobs along with a set of operator-level information about the instantiated jobs (i.e., job instances metadata).

At last, following an active learning approach, the *Label Forecaster* (Section 5) chooses few job instances through the *Job Execution Sampler* 3 to execute. At the same time, it computes the features of the job instances sample through the *Feature Extractor*. Next, through the *Model Builder*, it trains a predictive model with the features of the executed jobs 4a and the collected execution time 5. The *Forecaster* then exploits the features of the non-executed jobs 4b and the model 6a to predict the missing labels (i.e., runtime). The *Uncertainty Evaluator* computes the labels' uncertainty analyzing the model 6b and the forecasted values 7. While still highly uncertain, the *Job Execution Sampler* exploits the forecasted values to incrementally sample and execute a new small set of the most uncertain 8. Then, the Label Forecaster updates the predictive model and improves label quality. The output of the whole process is an augmented dataset of labeled jobs, tailored to the input query workload, input data, and computational resources 9.

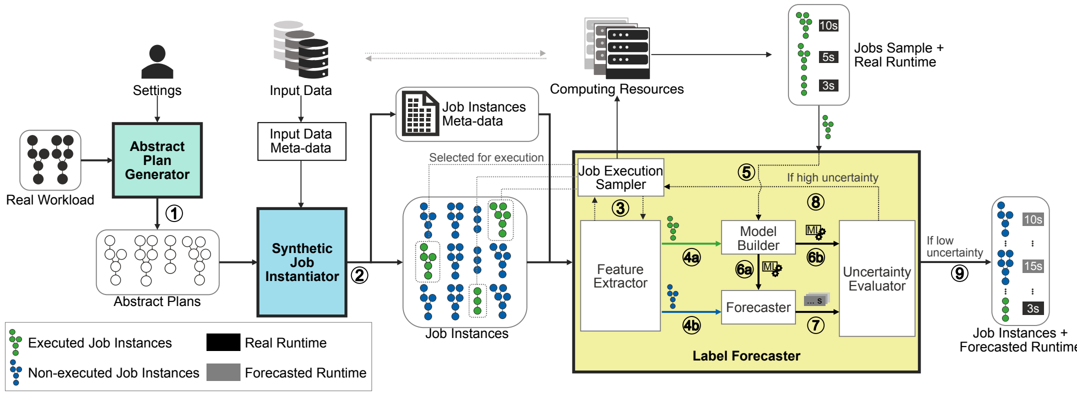


Figure 2: DataFarm's training data generation process. The process is composed of three main components: (i) Abstract Execution Plan Generator, (ii) Synthetic Job Instatiator, and (iii) Label Forecaster.

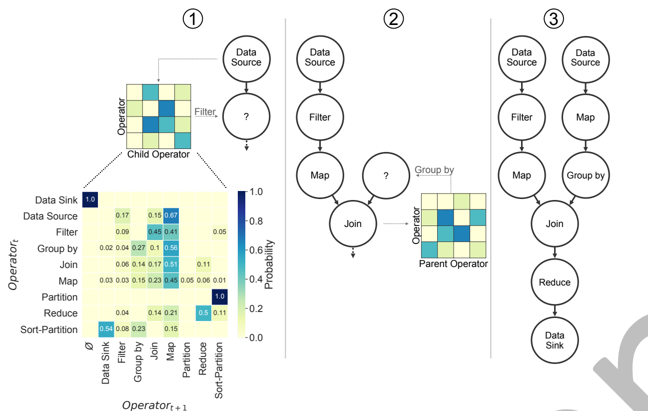


Figure 4: Abstract Plan generation with Transition Matrix.

We designed DataFarm in a way that one can easily extend (i) the *Abstract Plan Generator* to support new operators, (ii) the *Synthetic Job Instantiator* to support custom UDFs and input data, and (iii) the *Label Forecaster* to support different target labels, such as execution time or output cardinality. The current implementation supports the most common operators of Apache Flink [8], i.e., Source, Sink, Map, Reduce, Filter, Join, GroupBy, Partition, and SortPartition. We use DataFarm into Agora, a data infrastructure for AI sharing and innovation [49].

3 GENERATING ABSTRACT PLANS: IMITATING REAL WORKLOAD PATTERNS

In practice, it is common to have a workload containing few frequent execution patterns (e.g., many joins and few filters). One has to consider these patterns to build a reliable augmented workload.

> The *Abstract Plan Generator* (APG) addresses the problem of (i) learning execution patterns from a small real workload and (ii) generating new plans that are representative of the real ones.

The main goal of APG is to create new plans that follow the distribution of the operators found in the input workload, while increasing their variety. Following the distribution of input workloads prevent us from creating non-meaningful sequences of operators. In data management systems, jobs are usually defined by logical plans implemented as DAGs: the nodes represent unary or binary logical operators and the links describe the input/output relations among them. APG iteratively generates synthetic abstract plans taking as input a few existing logical plans and analyzing their structures and characteristics. The APG algorithm is composed of two main phases: the *learning* and *generation* phases.

Learning phase. First, APG learns general statistics about the input workload, such as the distribution of the longest path lengths and the number of joins in each plan. Then, to learn the common execution patterns contained in the real jobs, each logical plan is analyzed as a Markov Chain [15]. Each node in the DAG, i.e., each operator, represents a possible state of the system independent from the previous and the next one. Thus, APG learns two transition matrices [15], one for the previous and one for the following state. In this context, a transition matrix is a square matrix used to describe all the possible transitions from an operator to another. APG learns the probability of each possible transition by considering its relative frequency of appearance in each input plan. The first transition matrix, the *Children Transition Matrix* (*CTM*), contains the probability mass function of each operator o at step t describing its probability of transiting to any other *child* operator at step $t+1$ (see left bottom corner in Figure 4). *CTM* can be queried to get the probability $\mathbb{P}(o_{t+1}|o_t) = CTM_{o_t,o_{t+1}}$ of moving from a given operator o_t to a child operator o_{t+1}. The second transition matrix, the *Parent Transition Matrix* (*PTM*), contains the probability mass function of each operator o at step t describing its probability of transiting to any other *parent* operator at step $t-1$. *PTM* can be queried to get the probability $\mathbb{P}(o_{t-1}|o_t) = PTM_{o_t,o_{t-1}}$ of moving from a given operator o_t to a parent operator o_{t-1}.

Generation phase. The generation phase follows the distribution learned from the input workload. It takes advantage of the learned matrices to create abstract plans that imitate the structure and patterns of the already existing jobs. For each abstract plan, if not differently specified by the user, APG defines the maximum path length and the maximum number of joins for the new plan by sampling the corresponding distributions discussed in the learning phase. Figure 4 shows the main steps of this phase for our running example. Each new abstract plan starts with a `DataSource` node. Figure 4 (1) shows the beginning of the generation process. Then, the algorithm iteratively proceeds forward by sampling the probability mass function of the current operator CTM_{o_t} and deciding which would be the next operator o_{t+1} to insert in the abstract plan. As probabilities are used as weights when sampling, a transition having zero probability will never appear in the abstract plans.

Every time a `Join` operator is encountered, the algorithm starts a new backward branch. Figure 4 (2) shows the beginning of the backward generation process. From a `Join` operator, the algorithm iteratively proceeds backward by sampling the probability mass function PTM_{o_t} and selecting which would be the parent operator o_{t+1}. The backward generation proceeds until a new `DataSource` is encountered or if half of the maximum depth is reached. As a design choice, the algorithm does not allow for introducing further `Join` operators in backward branches, producing left deep plans only. This allows identifying, in the instantiation phase, the main execution branch that contains all the `Join` operators while all the other execution branches will just merge with the main one. Once a backward branch is complete, the forward generation continues from the lastly introduced `Join`. If the maximum number of `Join` operators is reached, we simply skip them when sampling the children operators. The generation process terminates when a `DataSink` is encountered or when the specified maximum depth is reached. Figure 4 (3) shows the generated abstract plan for our running example after the final step of the generation phase.

The generation process is repeated as many times as the number of required abstract plans θ. The output is a collection of abstract plans $P = \{p_0, \ldots, p_\theta\}$ that needs to be instantiated.

4 INSTANTIATING ABSTRACT PLANS: CONSIDERING REAL DATA DISTRIBUTION

The abstract plans $P = \{p_0, \ldots, p_\theta\}$ outputted by the abstract plan generator cannot be executed yet for two reasons. First, they are not concrete jobs for a specific system. Second, the operators' user-defined functions (UDFs), such as selection predicates, have not been instantiated. To generate a representative query workload, it is of primary importance to instantiate UDFs tailored to real data distributions. Managing the wide range of UDFs along with their parameters and the possible join orderings that can be implemented is the main challenge we face when instantiating abstracts jobs.

> The *Synthetic Job Instantiator* (SJI) addresses the problem of (i) creating an augmented set of executable jobs for each abstract plan, and (ii) including real input data metadata and custom UDFs in the instantiation process.

The *Synthetic Job Instantiator* (SJI) leverages statistical analysis of the input data and instantiates realistic jobs including custom UDFs. SJI also exploits input data's metadata to ensure a realistic instantiation for each operator making them always executable. For example, `Filter` operators exploit only fields and values that can be filtered, and `Joins` are performed only among joinable fields.

Given a set of abstract plans, SJI provides different heterogeneous job instances for each of them covering most of the possible relevant queries that can be performed with the input data. In the following, we first explain why input data metadata is necessary for the instantiation process and detail how we compute it (Section 4.1). We then discuss how the instantiation algorithm works (Section 4.2) and describe why and which job instances' metadata is collected during the instantiation process (Section 4.3).

4.1 Leveraging Input Data Metadata

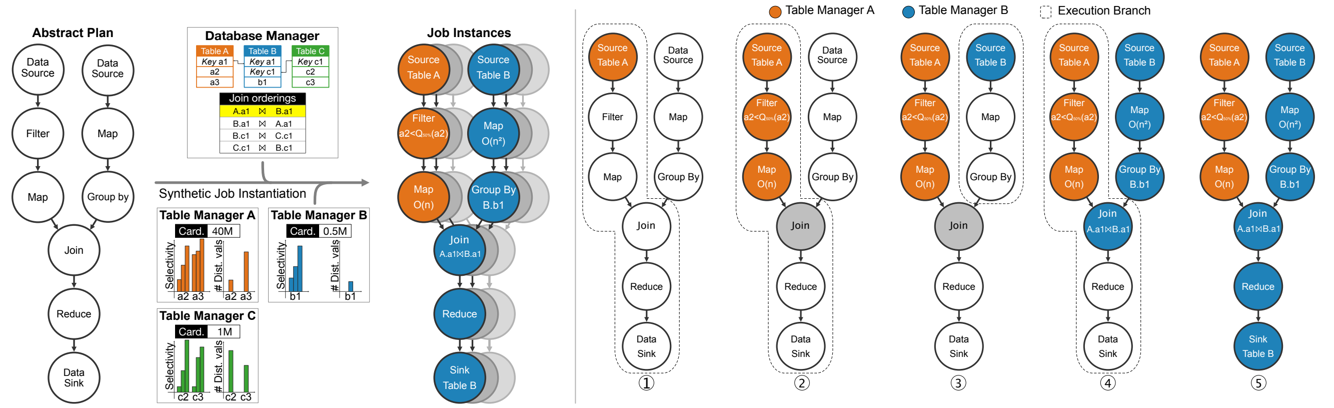
Besides the abstract plans, SJI also receives *input data metadata*, which is necessary to tailor jobs instances to each use case (see Figure 2). The input data metadata is composed mostly of structural information and statistics that add great value to the instantiation process. First, SJI has to be aware of the schema of the data: the fields that can be joined, and the ones that can be filtered and/or grouped. Second, it is necessary to extract statistics, i.e., table cardinalities, the number of distinct values for each field that can be grouped, and the discrete distribution of the values in each field that can be filtered. The metadata related to the input data schema is necessary to handle all the possible join orderings and the different combinations of input parameters while instantiating the operators. The metadata related to cardinality and values distribution has a two-fold function: (i) SJI exploits this metadata to decide the fields and the values to filter on or the fields to group by; and (ii) SJI keeps track of these decisions as jobs' metadata, e.g., filters selectivity or group by output cardinalities, to be able to characterize each instantiated job at the operator level. SJI offers two interfaces to integrate the input data metadata in the generation process: the (i) *Database Manager* and (ii) *Table Manager*.

The *Database Manager* (DM) is the interface that allows users to include a new database with its tables and their relations (Database Manager in Figure 5a). SJI requires the DM to be aware of the database schema to consider each possible join ordering. The *Table Manager* (TM) is the interface that allows to include per-table statistical metadata. Each table in the database has to be represented with an implementation of the TM interface (Table Managers in Figure 5a). Then, each TM implementation should include the information about the table cardinality, the discrete distributions of each filterable field, and the number of distinct values in each group by field. Note that a TM defines a particular instantiation strategy for an abstract operator and provides the necessary tools to implement new operators' UDF. Thus, with a minimum effort, one can provide a custom instantiation strategy for each logical operator. SJI provides a default TM implementation for most operators.

4.2 Instantiating Abstract Plans

The instantiation process takes as input a set of abstract plans (P), the number of instances (I) to generate for each $p \in P$, and the input data metadata represented by a *Database Manager* and a list of *Table Managers*. Figure 5a shows an example of inputs and outputs of the instantiation process for a given abstract plan and the input dataset described in Section 2.

For each input abstract plan p, SJI produces a collection of possible join orderings by leveraging the Database Manager. The join orderings are represented by table sequences, where the length of each table sequence matches the number of data sources in p. For each job instance $i \in I$, SJI chooses a new table sequence and starts a recursive instantiation of the execution branches (i.e. sequence of unary operators) contained in p. The main execution branch is the one selected in the first recursion going from source to sink. All the remaining ones are sub-branches that start from other sources



(a) (b)

Figure 5: Job instantiation running our example: (a) overview of Job instantiation given one abstract plan, the *Dataset Operator Manager*, and the *Table Managers*; (b) main steps of the job instantiation.

and reach the join operators merging with the main branch. Recall sub-branches do not contain any other join operator (see Section 3).

The recursive instantiation begins by extracting the main execution branch and the first Table Manager *tm* from the table sequence. The algorithm then iterates over the abstract operators contained in the execution branch and exploits *tm* to obtain a valid instance of the current operator. Figure 5b-① shows the instantiation of the first `DataSource` operator in the main execution branch. If a `Join` operator is encountered in the main execution branch (Figure 5b-②), the algorithm recursively starts the instantiation of the next sub-branch that converges to it. Thus, the visit of the new sub-branch begins from its `DataSource` (Figure 5b-③), the next *Table Manager* is selected, and the algorithm proceeds iteratively instantiating all the subsequent operators. Once consumed by all the operators in the sub-branch, the algorithm returns to the main branch updating the old *Table Manager* with the one in the sub-branch and instantiating the `Join` operator (Figure 5b-④). The job instantiation proceeds following the previous steps until the `DataSink` is reached. Figure 5b-⑤ shows a completed job instance.

SJI follows a pseudo-random approach to generate different job instances for the same abstract plan. When the operator instantiation requires selecting the input field or the input argument (e.g., in the case of groupby or filter operators) and its value, the corresponding Table Manager randomly chooses one of the possible fields and samples its values. The output of the instantiation process is a set of executable job instances $J = \{j_{0,0}, \ldots, j_{p,i}\}$ with $p \in P, i \in I$.

4.3 Characterizing Jobs with Metadata

During the instantiation process, all the decisions made by SJI are recorded to be used in the next steps. The *Job Instances Metadata* (see Figure 2) provides a detailed operator-level description of each instantiated job. We can use jobs' metadata to assess the content of each job enhancing the interpretability of the process and ensuring complete coverage of the input data. The jobs' metadata is also important to extract features for each job (as we will see in Section 5.2). The challenge is that real cardinalities are not available for the generated operators, apart from `Source` operators. The input data metadata includes only the raw cardinalities of the input tables and the selectivities of the filterable values on raw input data. We estimate all the other operator's cardinality using textbook estimation techniques [17].[3] SJI extracts the recorded metadata during instantiation from Table Managers, which contain information about the input distribution. In detail, we record: (i) the selectivity of each operator instance during the job instantiation; (ii) the UDF computation complexity for `Map`-like operators, and; (iii) the estimated output cardinality for `Groupby` and `Reduce`-like operators.

5 ACTIVE LABEL FORECASTING: LABELING QUERY WORKLOADS

Generating a large set of heterogeneous jobs is not sufficient for the new era of learned data management components. The runtime of these jobs is also crucial for training supervised ML models because it serves as the label of the training data. However, executing all this amount of jobs is impractical: We observed in our experiments that executing 10,000 OLAP queries using Flink [4, 8] on a 50GB dataset and a four-quadcore-nodes cluster takes more than a month. We, thus, label the augmented set of job instances exploiting an active learning approach. We iteratively label some of the jobs by actually executing them and forecast the labels of the non-executed jobs using an ML model.

> The *Label Forecaster* (LF) addresses the problem of (i) characterizing jobs by means of interpretable and representative features, (ii) finding the smallest possible set of representative jobs to execute, and (iii) predicting the labels along with uncertainty for the non-executed jobs.

The cost of executing a set of jobs is strictly related to the available computational resources and data management platforms. Defining a cost model that takes into account different platforms and the huge amount of parameters contributing to the runtime of a job is a complex, still unsolved, problem. Our *Label Forecaster* (LF) overcomes this complexity by inferring the task-specific cost model directly from the available data at the cost of collecting few labeled samples necessary for the model training. In the following, we first introduce and explain the active-learning-based strategy

[3] Cardinality estimation is out of the scope of this work.

Algorithm 1: Active labeling.

Input : jobs instances J; computational resources R; number of init. jobs κ; threshold η; early stopping threshold λ; max iterations MAX_i

Output : set of labels L

1 $L \leftarrow \emptyset$; $L_{ex} \leftarrow \emptyset$; $\hat{L}_{noex} \leftarrow \emptyset$; $J_{ex} \leftarrow \emptyset$; $J_{noex} \leftarrow \emptyset$; $U \leftarrow \emptyset$; $i \leftarrow 0$;
2 $F \leftarrow$ FeatureExtractor.transform(J);
3 $S_{ex} \leftarrow$ JobExecSampler.initialize(F, κ);
4 **while** *not earlyStop*(U, λ) *or* $i < MAX_i$ **do**
5 $\quad L_{ex} \leftarrow L_{ex} \cup R$.submit($S_{ex}$);
6 $\quad J_{ex} \leftarrow J_{ex} \cup S_{ex}$;
7 $\quad J_{noex} \leftarrow J \setminus J_{ex}$;
8 $\quad M \leftarrow$ ModelBuilder($F[J_{ex}]$, L_{ex});
9 $\quad \hat{L}_{noex} \leftarrow$ Forecaster($F[J_{noex}]$, M);
10 $\quad u \leftarrow$ UncertaintyEstimator($\hat{L}_{noex}$, M);
11 $\quad U \leftarrow U \cup \{u\}$;
12 $\quad S_{ex} \leftarrow$ JobExecSampler.nextExecs(J_{noex}, u, η);
13 $\quad i \leftarrow i + 1$;
14 **end**
15 $L \leftarrow L_{ex} \cup \hat{L}_{noex}$;
16 **return** L

(Section 5.1). We, next, describe the feature extraction process (Section 5.2) and present the details of the model building, forecasting, and uncertainty evaluation (Section 5.3). We then discuss how we sample jobs for execution (Section 5.4).

5.1 Active Learning Strategy

One of the main challenges with any data-driven solution is the cost of labeling samples, in our case the cost (in terms of money and time) to execute jobs. To tackle this problem, we follow an active learning approach [14] based on quantile regression forests [35]. An active learning strategy allows the LF to incrementally execute the unlabeled job instances by selecting only the most informative ones. Thus, the learner, iteration after iteration, executes only the jobs that really add new information to the ML model. This allows the LF to reach high predictive performance with fewer training labels w.r.t. traditional ML approaches.

Algorithm 1 describes the main steps of the active learning process used by LF (see also Figure 2). Initially, no labels are available in the set of jobs' labels L. From the input set of job instances J, LF computes the set of operator-level features F through the *Feature Extractor* (line 2). Then, through the *Job Execution Sampler*, LF selects the initial subset of the workload to execute, S_{ex}, exploiting an unsupervised analysis of F that maximizes the differences between the available jobs (line 3). Next, the selected jobs S_{ex} are submitted to the computing resources R, their execution times are collected in L_{ex} (line 5), and the sets of executed jobs J_{ex} and non-executed jobs J_{noex} are consequently updated. Via the *Model Builder*, LF exploits the labels extracted from the executed jobs L_{ex} and the corresponding features $F[J_{ex}]$ to build the predictive model M (line 8). At the same time, it predicts the set of labels $\hat{L}_{noex}$ given the features of the non-executed jobs $F[J_{noex}]$ and the trained model M through the *Forecaster* (line 9). As the labels are estimated, it is important also to evaluate the uncertainty of the predictive model in the forecasting process. LF computes the uncertainty u for each forecasted label in $\hat{L}_{noex}$ using the *Uncertainty Evaluator* (line 10). We discuss the details about the *Model Builder*, *Forecaster*, and *Uncertainty Evaluator* in Section 5.3. Before completing an iteration, LF selects the jobs with the highest uncertainty (line 12) to execute them and to update the predictive model increasing its performance in the next iteration. This iterative learning process between lines 4-14 is repeated until the model's uncertainty U matches a given early stopping condition λ or if the maximum number of iterations MAX_i is reached. The output of the algorithm is the union of the set of extracted labels and the set of forecasted labels (line 15).

5.2 Interpretable Operator-Level Features

One of the main steps of LF is the feature extraction from each job instance. We aim for an operator-level feature extraction process using statistical analysis. Recall that the job instances produced by the job instantiator have been characterized by operator-level metadata, which LF exploits when extracting features. Operator-level statistics has several advantages concerning plan-level features [3] producing complex embeddings [32] or using mixed approaches with embeddings at query- and plan-level [30]:

(1) Operator-level statistical features describe more precisely the characteristics of a job w.r.t. plan-level ones, including latent details about the data processed during the execution.

(2) Statistical features are more interpretable compared to complex embeddings leading to a white-box interpretable outcome.

(3) To build a robust embedding neural network, like the one proposed in [30], an onerous training phase is required while computing statistics is much more efficient.

The cardinality of the input data exploited by each job is one-hot-encoded setting the cardinality value if the table is used and zero otherwise. The UDF complexity of `Map` operators is represented by the order of magnitude of the number of fields in each record times the number of records analyzed by the `Map` instance. The order of magnitude is represented by powers of 10 by applying the corresponding complexity function $O(1)$, $O(n)$, or $O(n^2)$. We represent all other logical operators by statistics computed on the estimated output cardinalities. The more the output cardinality estimation for each operator is precise, the better the jobs will be characterized. LF exploits cardinality estimation based on the histograms provided in the input data metadata, however, any estimation technique can be applied at this step. Notice that output cardinality estimation is a very complex task [27, 46] extensively studied in the literature [18, 24, 34, 46] and it is out of the scope of this work.

Among the superset of features that could be extracted by analyzing the instantiated jobs, e.g., #operators, #joins, or max logical path length, we keep only the features with the most relevant variance. We do this by analyzing their principal components through PCA (Principal Component Analysis). Furthermore, we remove noisy features by checking their feature importance during the learning process, as defined for CART algorithms [7].

5.3 Model Building and Forecasting

Forecasting jobs' execution time is a supervised regression process and it requires training on labeled data. The ML model at the core of this learning process is the quantile regression forests [35] (QRF). QRF belongs to the family of Random Forest (RF) algorithms, known in general for their robustness and flexibility in learning very complex functions with high-dimensional data [6, 43]. Moreover, RFs are also considered *white-box* algorithms by explaining the performed predictions through feature importance analysis [7]. The standard RF, however, is limited by the approximation made while predicting output labels because it performs a weighted average among the decisions taken by each inner tree. Quantile Regression Forest, instead, overcomes this issue by estimating the decisional distribution function of the inner trees. Thus, through QRF, conditional quantiles about each outcome are extracted providing the prediction interval in which, with high probability, the forecasted value is going to be. In practice, QRF allows understanding how much the model is *uncertain* about the forecasting of execution times, further improving the interpretability and reliability of the proposed approach. Below, we detail the model building, forecasting, uncertainty evaluation, and learning early stopping heuristics.

Training and forecasting with uncertainty. The *Model Builder* trains and updates the QRF model at each iteration of Algorithm 1. The model is trained on the logarithm of the execution time of the set of executed jobs. This non-linear transformation allows for balancing the high disparity that usually exists between jobs' execution time (i.e., different jobs executed on the same dataset can take from a few milliseconds to hours), simplifying the learning process. The *Forecaster* exploits the trained QRF model to predict the labels for the set of non-executed jobs by applying the exponential operation to take back the estimated values to the time domain. The *uncertainty* of each forecasted label is computed by the *Uncertainty Evaluator*. To do so, it queries the QRF model and extracts the prediction interval that is composed of the first and the third quartiles (i.e., 25% and 75%) of the model's decisional distribution function. These quartiles correspond, respectively, to the lower and upper uncertainty boundaries. Thus, each forecasted label is characterized by the estimated value $\hat{l} \in \hat{L}_{noex}$ and its prediction interval $[u_{low}(\hat{l}), u_{high}(\hat{l})]$. The smaller is the amplitude of the prediction interval, the lower is the label's uncertainty given by $u(\hat{l}) = u_{high}(\hat{l}) - u_{low}(\hat{l})$. Figure 6 shows an example of forecasted values centered w.r.t. the mean of the predictions.

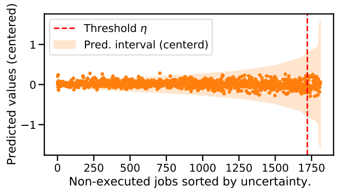


Figure 6: Predictions' uncertainty.

Assessing forecasting performance without ground truth. In our case, it is not possible to use standard supervised metrics for regression tasks, e.g., R^2 score [11], for estimating the quality of the forecasted labels of the non-executed jobs. This is because the ground truth (i.e., the real execution time) is not available. Also, evaluating the quality during the hyperparameter tuning phase of LF's model can lead to very unstable results because the number of executed jobs would be very low compared to the non-executed ones. Thus, LF analyzes the model's uncertainty itself. It calculates the model's *uncertainty* $\bar{u}$ by averaging all labels' uncertainties:

$$\bar{u} = \frac{\sum_{\hat{l}}^{\hat{L}_{noex}} u_{high}(\hat{l}) - u_{low}(\hat{l})}{|\hat{L}_{noex}|} \tag{1}$$

The intuition is that a decrease in the model's uncertainty during the iterations of Algorithm 1 represents a more accurate prediction. Thus, an increment of uncertainty can be a symptom of less reliable forecasting. In this sense, the model's *uncertainty* is a metric that can be exploited to evaluate the reliability and quality of the predicted labels. Experimental results show also that while the model's uncertainty decrease, the percentage of variance explained by the model, i.e., R^2 score, increases confirming that the model's uncertainty is an effective metric to measure the predictive performance in absence of ground truth values (see Section 6.6).

Early stopping active learning. Exploiting the analysis of the model's uncertainty, we use a heuristic to stop the incremental learning process. The model's uncertainty shows the tendency to decrease while incrementing the number of executed jobs: The more jobs are executed, the lower the uncertainty is and the highest the model's performance is supposed to be. However, we must find a compromise between predictive performance and the number of executed jobs, because every new iteration is potentially very costly due to the jobs' execution time. Taking a conservative approach on the number of executed jobs, we can stop the learning process every time the model's uncertainty shows a significant drop. That is, we can stop the process whenever the uncertainty is reduced more than a specific percentage λ w.r.t. the previous iteration. In addition, the current uncertainty has to be lower than all the previous ones.

5.4 Job Execution Sampler

Recall that the *Job Execution Sampler* has the role of querying the set of instantiated jobs and selecting only the most representative ones to be executed. A job can be representative in two different ways: (i) if it shares similar characteristics with a larger group of other jobs, and (ii) if its execution can reduce significantly the model's uncertainty. Furthermore, the set of selected jobs to be executed has to be as small as possible because submitting jobs can be very expensive and we want to avoid unnecessary runs. The Job Execution Sampler follows a completely unsupervised approach in that it does not require prior knowledge about jobs' execution time. It participates in two phases of the labeling forecasting: in the initialization phase (line 3 in Algorithm 1), and the model refinement phase (line 12), which we explain below.

Initialization sampling. At the beginning of Algorithm 1, no labels are available for any of the instantiated jobs. Users have to provide the number of jobs κ that has to be sampled in the initialization phase (a value of $\kappa \geq 50$ is empirically suggested). The initial set of jobs to execute is composed of two subsets extracted combining two different techniques.

The Job Execution Sampler selects the first subset of jobs by analyzing the logarithm of the jobs' *Source Card. Sum* feature. This feature characterizes each job with the sum of the cardinalities of their `Source` operators. We want to sample $\frac{\kappa}{2}$ jobs that equally represent jobs with small, medium, and high input cardinality. Thus, we first sort jobs by their *Source Card. Sum*. Then, we split the range of logarithm of *Source Card. Sum* values in $\frac{\kappa}{2}$ uniform intervals and select the closest job to the boundaries of each interval. This leads to a uniform coverage of the whole distribution of input cardinalities.

The Job Execution Sampler selects the second subset of jobs by analyzing the whole set of features available for each job. It exploits the Principal Component Analysis [51] (PCA) to extract a small set of principal components from the jobs' features. PCA is usually used to reduce the size of high-dimensional problems by aggregating correlated features and projecting them into a lower-dimensional space allowing to highlight latent patterns in the data. Then, the Job Execution Sampler identifies $\frac{\kappa}{2}$ groups of jobs through agglomerative clustering [41].

The clustering algorithm groups together jobs with correlated characteristics while keeping separate the uncorrelated ones. At last, the Job Execution Sampler selects the centroids of the clusters as a sample of jobs to execute because they show average characteristics among all the other jobs belonging to the same groups. This allows for executing jobs with maximum inter-differences.

Model refinement sampling. During the iterative model refinement process, the model is continuously trained with an increasing number of executed jobs. To select the next set of jobs to execute, the Job Execution Sampler analyzes the uncertainty of each forecasted label. It sorts the non-executed jobs J_{noex} on the labels' uncertainty $u(\hat{l})$ and selects the last η percent of them to be executed in the next iteration of Algorithm 1. Figure 6 shows an example of a threshold above which the $\eta = 5\%$ percent of most uncertain jobs are selected to be executed in the next iteration. The smaller η is, the finest is the job selection but the larger the number of required iterations to reach the convergence. This approach allows the model to refine its learning process in the next iterations by analyzing the jobs that caused a high level of uncertainty in the current step.

6 EXPERIMENTAL VALIDATION

The goal of DataFarm is to ease the use of ML in data management systems by providing suitably large labeled training data. Evaluating our framework is a challenge by itself as there is no available ground truth, i.e., large query workload with its labels. For this reason, we choose to validate DataFarm by showing (i) the representativeness of the generated jobs along with the quality of the estimated labels in Section 6.3 (ii) the effectiveness and the efficiency of the generation process in Section 6.4, (iii) a detailed comparison of our framework w.r.t. the current state-of-the-art in Section 6.5, and (iv) an in-depth analysis in Section 6.6. We have released DataFarm as open-source.[4]

6.1 Setup

We first elaborate on the different inputs and metrics we used.

Cluster setup. We ran all of our experiments using Apache Flink [4, 8] version 1.10.0 with the default settings on a four-node cluster. Each node is equipped with a quad-core Intel Xeon 2.40GHz CPU and configured with 16GB of main memory for each Flink worker.

Datasets and queries. We used both synthetic and real datasets together with a small input query workload from TPC-H. Specifically, we used (i) four TPC-H datasets generated with different scale factors, i.e., 1GB, 5GB, 10GB, 50GB, and (ii) the publicly available IMDB database[5]. The four TCP-H datasets are exploited to show that DataFarm is robust to high heterogeneity in tables cardinality and it can thus deal with both short- and long-running jobs in the same query workload. The IMDB dataset, instead, represents a typical real database with unbalanced data distributions. We exploit it to test the robustness and the generality of the proposed framework even in such a case. We used 6 TPC-H queries, i.e., Q1, Q3, Q11, Q13, Q17, and Q21, which include a mix of common execution patterns (i.e., pipelines, jobs with multiple joins and filters, groupby aggregations, materialized views), being representative of the entire TPC-H workload. We implemented them using the Flink DataSet[6] and Table[7] APIs resulting in 12 Flink jobs. This very small set of jobs constitutes the input to DataFarm. It includes all the operators currently supported by our generator, i.e., Sink, Map, Reduce, Filter, Join, GroupBy, Partition, and SortPartition.

Label Forecaster settings and validation metrics. We configured the active label forecasting process with (i) a maximum number of iterations $MAX_i = 20$, (ii) threshold η of 0.05 meaning that the 5% of the jobs with maximum uncertainty are executed in each successive iteration, (iii) the number of jobs to execute during the initialization κ set to 51, and, (iv) an early stopping condition λ of 10% of the model's uncertainty reduction. We used grid-search with 3-fold cross-validation for the hyperparameters tuning of the Label Forecaster (e.g., number of trees in Quantile Random Forest). We experimentally picked these parameters intending to find a good trade-off between the quality of the predicted labels and the time required for the training data generation. In detail: The larger the number of initial jobs is, the more information can be learned in the firsts iterations in exchange for execution time; Increasing η can reduce the efficiency of the process because more jobs are executed in every iteration and the best stopping condition may be located in between two iterations; Reducing η increases the number of iterations and the overhead for model training; Finally, MAX_i has to be set according to the user's maximum time budget and λ has to be small enough to find a stop before MAX_i is reached.

We used the R^2 scores [11] between the forecasted labels and the ground-truth execution times to validate the label forecaster. The R^2 score is a metric widely exploited in regression problems. It measures the proportion of variance of the ground truth that has been explained by the values forecasted by a model. The closer R^2 is to 1, the better the model performance is. To obtain the ground-truth values, we ran all the generated jobs to get their labels.

6.2 Generated Query Workloads with Labels

For the evaluation, we generated two workloads with labels:

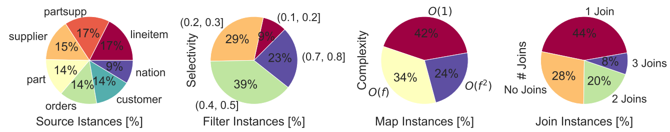
- W1: Our framework generates 2, 000 synthetic jobs with labels by analyzing our 12 Flink jobs and exploiting the four synthetic TPC-H datasets as input data. Thus, for each of the input datasets, it generates 50 abstract plans (with a maximum of 3 Join operators and a maximum branch depth of 6 operators) and it instantiates 10 different versions for each of them.
- W2: Our framework generates 1, 000 synthetic jobs with labels by using the transition matrices learned from the input-workload of W1 and exploiting the IMDB dataset as input data. Thus, it generates 50 different abstract plans (with a maximum of 3 Join operators and a maximum branch depth of 6 operators) and it instantiates 20 different versions for each of them.

6.3 Quality of Generated Query Workload

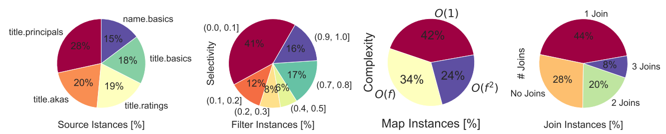
We first assess the quality of the generated query workload by evaluating both (i) the representativeness of the generated jobs and (ii) the accuracy of predicted labels.

Representativeness of jobs. To allow an ML-based optimizer to properly generalize and be robust to eventual outliers the training

[4]https://github.com/agora-ecosystem/data-farm
[5]https://www.kaggle.com/ashirwadsangwan/imdb-dataset
[6]https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/batch/
[7]https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/



(a) W1



(b) W2

Figure 7: Characteristics of generated query workloads.

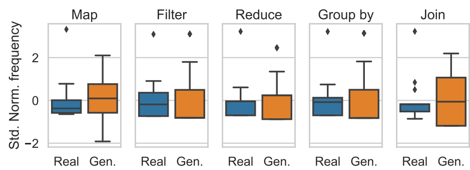


Figure 8: Normalized operators' distribution.

query workload has to be relevant to each use case: It should cover as much as possible the features of the input workload and data.

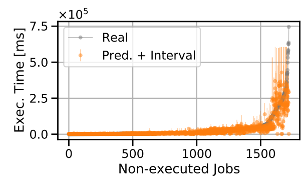
Figure 7 shows the distributions of various characteristics for the generated W1 and W2. We observe that DataFarm correctly distributes the available data sources among the generated jobs: All the input data is equally represented in the newly generated workload W1 and W2 (`Source Instances` charts in Figure 7). It is important to have a uniform representation of all the available tables since a generated query workload has to allow ML models to generalize in the learning process and to be robust in the prediction process. Then, we observe that our framework reflects the distribution of the data for other operators, e.g., the `Filter`, `Map`, and `Join` instances in Figure 7. In particular, we observe that the selectivity included in the `Filter` instances cover the possible input values exhaustively representing their distributions: W1 shows selectivity values almost uniformly distributed, while W2 selectivity is concentrated in range (0.0, 0.1] in 41% of the cases, as real input data distributions are skewed. Similarly, this holds for the UDFs complexity of `Map` operators and the number of `Join` operators. Note that the distributions of `Map` and `Join` operators are similar for W1 and W2 because they have been generated from the same input workload, i.e., from the same transition matrices. We observed the same patterns for all operators in the generated jobs.

These results reflect the power of DataFarm in capturing the input data distribution in the generated jobs and confirm the representativeness of the generated jobs for the user's use case. To further demonstrate this, we ran another experiment to study how well our generated workloads cover the user's workload.

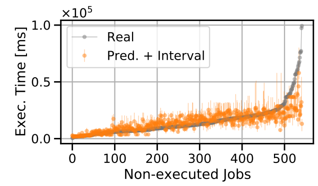
Figure 8 shows how the normalized operators' distributions in the real query workload compared with the generated ones. We observe that the operators' distribution in our generated workload has a similar mean to the operators' distributions in the real workload. We also see that the first and third quartiles of the operators' distribution in our generated workload always cover the ones of the operators' distributions in the real one. This shows not only the representativeness of our generated workloads but also its coverage

Table 1: Summary of generated workloads.

Generated Workload	# Gen. Jobs	# Exec. Jobs	Non-Exec. Jobs [%]	R^2 (Our)	R^2 (TDGen)
W1	2,000	142	93	**0.67**	<0
		532	73	**0.75**	0.65
W2	1,000	141	86	**0.16**	0.07
		418	58	**0.52**	0.08



(a) W1 – 142 executed jobs.



(b) W2 – 418 executed jobs

Figure 9: DataFarm forecasted labels with uncertainties.

of real workloads. We also observe that our generated workloads extend real ones with jobs that cover a wider range of cases.

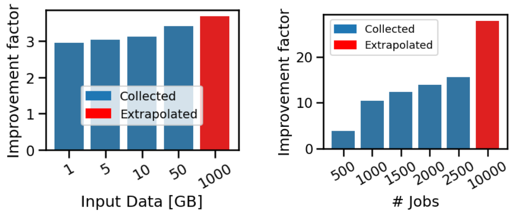
Accuracy of labels. Estimating the runtimes of generated jobs is necessary to get the labels to complete the training data. However, as we have already pointed out, executing all generated jobs can take weeks or even months and cost a lot of money, e.g., cloud provisioning costs. This is why our framework predicts the labels by executing only a few jobs from the generated query workload.

To evaluate how realistic the predicted labels are, we measure the R^2 score. Table 1 summarizes the results obtained for W1 and W2 on the first two early stoppings. For W1, the performance of the label forecasting process reaches $R^2 = 0.67$ by executing only 142 jobs. This means that most of the variance of the real execution times has been captured effectively by just executing less than 10% of generated jobs. For W2, the first early stop proposed by the algorithm reaches 16% of explained variance with 141 executed jobs. Instead, if the user has more time, on the second early stop it reaches an R^2 score of 0.52 and 418 executed jobs.

The quality of the predicted labels for W1 can be shown even more clearly by plotting them as a function of the executed jobs shown in Figure 9a. It plots the predicted labels for each one of the 1858 non-executed jobs along with their prediction intervals on a time scale. We observe that, independently of the size of the input data, almost all predicted labels, along with their prediction intervals, show values very close to the real execution time. Considering jobs with execution times reaching more than 12 minutes, the predictions show a very good median error of just 2 seconds while 75% of the forecasted labels remain below an error of 8 seconds.

We observe that the predicted labels have a very low uncertainty in short-running jobs while the prediction interval increases for long-running ones. This allows us to easily identify which are the generated jobs with possibly very long execution times and hence increasing the reliability of the proposed labels is important.

We observe similar results for workload W2 as shown in Figure 9b. In this case, the unbalanced distributions of the values in the input data are reflected in the generated jobs by the increased complexity in forecasting labels as shown in Figure 9b. From these results, we can see that to reach an amount of explained variance around 50%, a higher amount of executed jobs are required w.r.t the previous workload that is constructed from synthetic input data.



(a) Increasing input data. (b) Increasing number of jobs.

Figure 11: Effectiveness and efficiency of the framework.

6.4 Effectiveness and Efficiency

We now show how effective and efficient the generated training data is when used for an ML-based query optimizer that requires a query workload with execution times. We compare DataFarm with the common approach of manually labeling a query workload, i.e., by executing every single query in the workload.

To evaluate this, we use our generated jobs with both the *p*redicted and *r*eal runtimes as two training datasets to build an ML model that can be used by a query optimizer to predict runtime. It is worth noting that with an initial workload of 24 jobs (6 TPC-H queries with 4 different datasets), and thus only 24 labeled samples, it would be impossible to train an ML model.

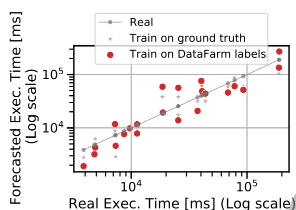


Figure 10: Effectiveness.

Effectiveness. We first evaluate the quality of our predicted labels by training two Random Forest regressors for W1: one with our predicted labels and one with the real ones (ground truth). We then predict the runtimes of our input query workload (i.e., TPC-H queries Q1, Q3, Q11, Q13, Q17, and Q21) using each model.

We observe in Figure 10 that the model trained on the generated workload with our predicted labels reaches a very good 73% of explained variance. This confirms once again that the proposed generation process is able to capture many relevant aspects of the input. Besides, it shows that having a limited number of mislabeled jobs does not significantly affect DataFarm's effectiveness: The model trained with ground-truth labels is only 1% more accurate, reaching just 74% of explained variance.

Efficiency. We now evaluate how much time users can save when using DataFarm instead of executing each query to obtain its runtime. We exploit the 500 jobs executed on four different scale factors in W1. In addition, we generate 500, 1,000, 1,500, 2,000, 2,500 jobs using the transition matrices learned from W1 and execute them on 1GB scale factor. We consider only the runtime of the Label Forecaster, as it is the computationally most expensive component: While the Abstract Plan Generator and the Synthetic Job Instantiator run in the order of seconds (e.g., 70*s* and 129*s* for generating 1, 000 and 2, 000 executable jobs, respectively), the Label Forecaster runs in the order of hours (e.g., 4*h* for W1 to execute 142 jobs).

Figures 11a and 11b show the results. We observe that DataFarm provides a linearly increasing improvement factor w.r.t. both the size of the input data reaching 3.4× (Figure 11a) and the number of executed jobs reaching up to 16× (Figure 11b). Interestingly, it achieves so without losing precision, i.e., the forecasted labels' explained variance ranges between 54% and 78%. Thus, the total improvement factor is about 54× for 2,500 jobs with 50GB input data ($3.4 * 16$). We also extrapolate the improvement factor for 1TB with 500 jobs and for 10,000 jobs with 1GB, leading to 3.7× and 28×, respectively (the red bars). This indicates that DataFarm's improvement factor increases significantly with the workload size. For example, consider we need to generate a training workload of 10, 000 jobs, which is a common size for training datasets [12, 30, 56], with an input data of 1TB. Thus, in this case, our framework reaches an improvement factor of 104× ($3.7 * 28$): It runs in less than 2 days while the current practice would require more than 6 months. DataFarm achieves such a performance gain because it always runs almost the same number of jobs. The number of executed jobs does not change as long as they cover the variance of the non-executed jobs and there is no concept drift [9, 16, 20, 22].

All these numbers show the high efficiency of our framework, making it possible to generate training data with thousands of heterogeneous queries while saving a huge amount of time.

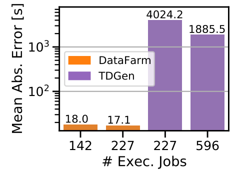
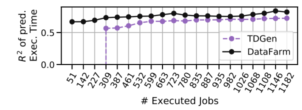
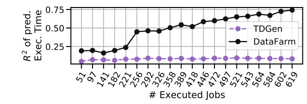


Figure 12: DataFarm vs. TDGen.



(a) W1



(b) W2

Figure 13: Labeling effectiveness w.r.t. TDGen.

6.5 Comparison with State-of-the-Art

We now compare DataFarm with TDGen [21], the only training data generator for ML-based query optimization that we are aware of. We generate a new set of 2, 000 jobs using TDGen and train the ML model presented in Section 6.4. We then evaluate the mean absolute error in estimating the costs of the real workload when training the model with DataFarm jobs and with TDGen jobs. Figure 12 shows the results. It is clear that DataFarm outperforms TDGen for more than two orders of magnitude: When we execute 227 jobs and predict the performance of the rest, the ML model's error is up to 236× lower when trained with DataFarm jobs (17.1*s*) than when trained with TDGen jobs (4, 024.2*s*). In fact, TDGen originally executed 596 jobs, and still, it had a large error (1, 885*s*).

This shows the importance of generating plans based on a small initial workload rather than on completely random jobs: TDGen does not take into account either users' real input workload or users' real input data. On the contrary, our framework is based on a completely different, data-driven approach that allows us to include both a small pre-existing workload and input data in the generation process. Next, due to these fundamental differences, we set a common ground by using the jobs generated by our framework and compare their effectiveness only in labeling a given query workload (*labeling effectiveness*). Lastly, we study the benefits of taking into account users input.

Labeling effectiveness. We use workloads W1 and W2 both generated by DataFarm and forecast their labels using TDGen and our active labeling approach. We measured the quality of the forecasted labels by exploiting the percentage of explained variance, i.e., R^2 score, for both W1 (Figure 13a) and W2 (Figure 13b).

Figure 13a shows that DataFarm can explain almost 70% of the variance of the real execution times already at the first iteration, i.e., executing just 51 jobs. In contrast, TDGen is much less efficient requiring to execute 309 jobs to reach a significant performance, more than 6 times the number of jobs required by our method.

Even more significant is the case of W2, where jobs have been generated with real input data. Figure 13b clearly shows that TDGen is not able to get close to the performance of our framework, even when executing more queries. This is because real input data (such as IMDB) is typically characterized by unbalanced value distributions. Thus, both DataFarm and TDGen show a low percentage of explained variance in the first iteration, even if our framework is already reaching better results than TDGen. DataFarm incrementally improves the prediction performance along with the number of executed queries: It is on average 6× more accurate while reaching up to 9× better prediction performance in the last iteration w.r.t. TDGen. While TDGen remains below 10% of explained variance.

Input data's importance. In contrast with TDGen that uses a completely random approach to generate jobs, the high effectiveness of DataFarm partly comes from the fact that it considers the characteristics of input data. To better evaluate this, we ran an experiment with a query workload of 2,000 jobs, which we generated using the W2's input queries but imposing a uniform data distribution. We then build a Quantile Regression Forest model taking this newly generated workload as training data along with its real labels to evaluate the best-case scenario. We used this model to predict the execution time for W2. Figure 14 shows the results. We observe that the predicted labels are completely skewed. It is clear that predicting execution times for W2, learning from a workload executed on a uniform input data distribution, is not effective. In fact, the result shows a negative $R^2 = -0.40$. This means that this model has worse performance than always predicting the average value of the ground-truth.

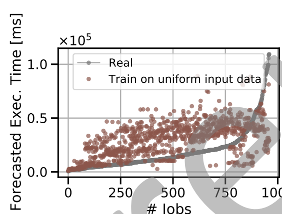


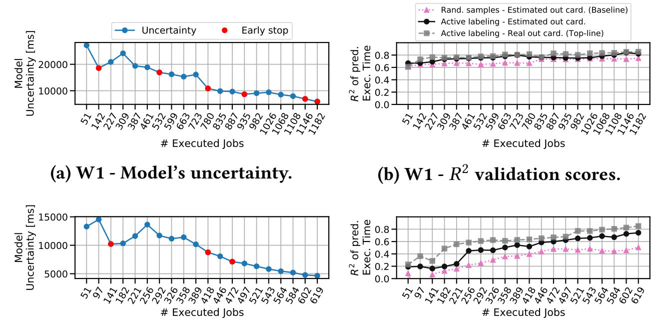
Figure 14: Data importance.

These results demonstrate the high complexity of the addressed problem while showing that our approach is a step forward to efficiently generate training data for ML-based query optimization.

6.6 In-depth Analysis

At last, we analyze different design choices of DataFarm. Figure 15 shows the model's uncertainty during the active labeling process and the validation scores computed to validate the obtained results. In the next paragraphs, the effectiveness of exploiting the model's uncertainty as a soft quality metric, of the active labeling sampling, and of the cardinality estimation will be discussed.

Effect of model's uncertainty. Our framework uses the model's uncertainty to determine when the labels obtained so far are sufficient and thus, to stop the active learning process. The results, in Figure 15, confirm that a decrement in uncertainty corresponds, in most of the cases, to an increment of the predictive performance. For instance, the overall trends of the model's uncertainty in Figures 15a and 15c are aligned with the trends of the R^2 scores for the active labeling approach shown in Figures 15b and 15d for W1 and W2. In other words, if the uncertainty decreases overall, the percentage of explained variance tends to increase.



(a) W1 - Model's uncertainty.

(b) W1 - R^2 validation scores.

(c) W2 - Model's uncertainty.

(d) W2 - R^2 validation scores.

Figure 15: Labeling model's uncertainty and R^2 scores.

Effect of active labeling. Thanks to the data-driven strategy we use in the Job Execution Sampler, the proposed active labeling outperforms the baseline of randomly sampling jobs. The comparison between active labeling and random sampling is shown in Figures 15b and 15d for W1 and W2. The active labeling always achieves better R^2 scores than the random sampling, even from the very first iterations where the number of labeled samples is very low and the complexity of the learning process is very high. Also, our approach reaches higher explained variance, i.e., R^2, with less executed queries. This means that both the initialization (first iteration) and the model refinement phase (successive iterations) are efficient strategies for selecting the set of queries to execute. It achieves high performance already from the first iteration and increases it successively. We also studied using a weighted random sampling strategy of the jobs to execute, with jobs' uncertainties as weights, instead of sampling based on top uncertain jobs. We measured the mean absolute error of the ML model built for a query optimizer (as in Section 6.4). In this case, sampling top uncertain jobs rather than using weighted sampling, reduced the error of the ML model of 1.7×, i.e., from 29.7s to 18.0s by running 142 jobs. This is because weighted sampling leads to the execution of also correctly labeled jobs. However, running these jobs is lowering the learning speed of the Label Forecaster and results in lower accuracy.

Effect of outliers. We now study the effects of outliers, e.g., caused by computing resources contention, during the labeling process. We consider the second labeling iteration for W1, i.e., 142 executed jobs in Figure 15a and synthetically add a random overhead of execution time (up to an increment of 100%) to an increasingly larger subsample of executed jobs (i.e., 0%, 10%, 25%). We, then, train an ML model with the workloads containing outliers and validate each resulting model as explained in Section 6.4. For this, we evaluate the mean absolute error (Figure 16). The results show that

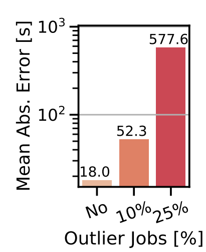


Figure 16: Outliers.

interference during the labeling process can lead to increasingly larger errors in the ML model. However, note that a manual collection of labels would also be affected by outliers. Besides, we observe that the uncertainty computed by the Label Forecaster increases up to three orders of magnitudes with 25% of outliers. This confirms that the proposed uncertainty metric correctly reflects the model's performance.

Effect of cardinality estimation. Recall the label forecasting process computes features from the estimated output cardinalities. We now evaluate how these estimations affect the predictive performance of the ML model and thus, the estimated labels. To achieve this, we perform the active labeling experiments with the real cardinalities, which we have extracted using the Flink task manager when each job is running. The results in Figures 15b and 15d show that a basic cardinality estimation process affects only marginally the learning process. For W1, the predictive performance obtained with estimated and real operators' cardinalities are comparable, showing an average difference in explained variance of just 3%. Instead, for W2 (Figure 15d) the real cardinalities allow active labeling to be more efficient by reaching 50% of explained variance with 182 executed jobs, while with estimated cardinalities only 20% is reached. However, this further demonstrates that our DataFarm is effective even with errors in the cardinality estimation process. As the cardinality estimation is still an open problem, we expect that by using more accurate estimation techniques the quality of the label estimation will improve significantly. However, investigating cardinality estimation methods is out of the scope of our paper.

6.7 General applicability

We have designed DataFarm to be task-, platform-, and hardware-independent. One can extend the *Abstract Plan Generator* and the *Synthetic Job Instantiator* to support any type of operator, data type, and UDF by simply implementing the provided interfaces as well as to instantiate jobs on any big data platform (e.g., Flink [8], Spark [54] or Rheem [1, 2]). In addition, the *Label Forecaster* is completely data-driven and does not rely on the underlying hardware. In our case, it collects the runtimes of the generated executed jobs and, based on this, it learns to predict the runtime of the non-executed ones regardless of the hardware used (e.g., CPUs or GPUs). Finally, DataFarm's generation process can be customized to work for other target labels of the generated jobs. The only component affected is the *Label Forecaster*. For instance, instead of collecting the runtimes of executed jobs, it can collect the output cardinalities during the active labeling. In this way, it can generate training data for learned cardinality estimation components [18, 23]. Figure 17 shows preliminary results of using DataFarm to label W2 with output cardinalities instead of execution time. This shows the extensibility of our approach.

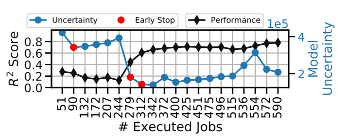


Figure 17: Cardinalities labels.

7 RELATED WORK

Data augmentation and dataset generation techniques with weak supervision have been exploited in many ML domains [39, 40, 42, 45, 50]. However, just a few attempts have been done in the database community. Most of the ML-based works in the database literature, often rely on task-specific benchmarking workload generators, such as TPC-H [37] or TPC-DS [36]. However, these tools are thought of as benchmarks: they generate synthetic workload starting from fixed execution patterns and, more importantly, they do not provide any labels, still requiring to execute a large workload to get them.

A first attempt to synthetic workload generation with labels has been recently proposed in [21]. In this work, the authors exploited a heuristic approach to generate new query workloads and estimating labels, i.e., jobs' execution times: They execute a large number of short-running jobs and only a few long-running ones and then interpolate the real extracted values to estimate the labels for the rest of the workload. However, [21] does not consider real input workload and data, which impacts the model's efficiency as we observed in our experimental comparison with this work (TDGen).

Other works facing the problem of lack of labeled training samples are related to active learning approaches [14, 44]. Active learning is a common strategy employed in ML [5, 13, 28, 48] that has been recently applied to data management problems as well [29]. In the latter, the authors address the performance degradation of ML-based data management when incoming data differs from data used to train the initial model. However, active learning alone does not solve the lack of training samples. , i.e., the availability of heterogeneous query workload.

Many works address also the problem of predicting query workload performance exploiting ML algorithms [3, 21, 30, 33, 38, 47, 52, 55]. However, these works usually train and validate their solutions on synthetic query workloads or on real workloads by manually collecting ground-truth labels. Moreover, the solutions that rely on deep learning approaches [30, 33, 47, 55] are not meant to be used on small data. Thus, they are not suitable to efficiently estimate labels by learning from few training samples, such as in DataFarm. Other approaches do not provide the models' uncertainty, e.g., [3, 38], and thus cannot be used to decide which are the jobs to execute in our active learning approach.

8 CONCLUSION

We presented an innovative query workload augmentation framework with efficient labeling estimation. DataFarm aims at fulfilling the increasing need for large training data to train ML-based data management components. It allows users to generate a large number of synthetic jobs by (i) imitating the execution patterns of a small pre-existing real query workload, (ii) tailoring job instances to the input data making them representative of each use case, and (iii) estimating labels for each generated job by actively learning their performance. We validated the quality of the generated jobs with an extensive experimental evaluation. The results showed that DataFarm outperforms the state-of-the-art in both effectiveness and efficiency. Moreover, we demonstrated that it allows users to save 54× (and up to an estimated 104×) time compared to the common approach of executing all queries.

Acknowledgments. This work was funded by the German Ministry for Education and Research as BIFOLD – Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).

REFERENCES

[1] Divy Agrawal, Mouhamadou Lamine Ba, Laure Berti-Équille, Sanjay Chawla, Ahmed K. Elmagarmid, Hossam Hammady, Yasser Idris, Zoi Kaoudi, Zuhair Khayyat, Sebastian Kruse, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Mohammed J. Zaki. 2016. Rheem: Enabling Multi-Platform Task Execution. In *SIGMOD*. 2069–2072.

[2] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed K. Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. 2018. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -. *Proc. VLDB Endow.* 11, 11 (2018), 1414–1427.

[3] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 390–401.

[4] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* 23, 6 (Dec. 2014), 939–964. https://doi.org/10.1007/s00778-014-0357-y

[5] Arvind Arasu, Michaela Götz, and Raghav Kaushik. 2010. On Active Learning of Record Matching Packages. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 783–794. https://doi.org/10.1145/1807167.1807252

[6] Gérard Biau, Luc Devroye, and Gábor Lugosi. 2008. Consistency of Random Forests and Other Averaging Classifiers. *J. Mach. Learn. Res.* 9 (June 2008), 2015–2033.

[7] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32. https://doi.org/10.1023/a:1010933404324

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[9] Tania Cerquitelli, Stefano Proto, Francesco Ventura, Daniele Apiletti, and Elena Baralis. 2019. Towards a Real-time Unsupervised Estimation of Predictive Model Degradation. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, BIRTE 2019, Los Angeles, CA, USA, August 26, 2019*. 5:1–5:6. https://doi.org/10.1145/3350489.3350494

[10] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1241–1258. https://doi.org/10.1145/3299869.3324957

[11] Norman Richard Draper and Harry Smith. 1998. *Applied Regression Analysis* (3rd ed ed.). Wiley, New York.

[12] Mark Everingham, S. M. Ali Eslami, Luc Van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. 2015. The Pascal Visual Object Classes Challenge: A Retrospective. , 98–136 pages. https://doi.org/10.1007/s11263-014-0733-5

[13] Weijie Fu, Meng Wang, Shijie Hao, and Xindong Wu. 2018. Scalable Active Learning by Approximated Error Reduction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 1396–1405. https://doi.org/10.1145/3219819.3219954

[14] Yifan Fu, Xingquan Zhu, and Bin Li. 2013. A Survey on Instance Selection for Active Learning. *Knowledge and information systems* 35, 2 (2013), 249–283. https://doi.org/10.1007/s10115-012-0507-8

[15] Paul A. Gagniuc. 2017. *Markov Chains: from Theory to Implementation and Experimentation.* John Wiley & Sons, Hoboken, NJ.

[16] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A Survey on Concept Drift Adaptation. *ACM Comput. Surv.* 46, 4, Article 44 (March 2014), 37 pages. https://doi.org/10.1145/2523813

[17] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2 ed.). Prentice Hall Press, USA.

[18] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1035–1050. https://doi.org/10.1145/3318464.3389741

[19] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, Not from Queries! *Proc. VLDB Endow.* 13, 7 (March 2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[20] Geoff Hulten, Laurie Spencer, and Pedro Domingos. 2001. Mining Time-Changing Data Streams. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California) *(KDD '01)*. Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/502512.502529

[21] Z. Kaoudi, J. Quiané-Ruiz, B. Contreras-Rojas, R. Pardo-Meza, A. Troudi, and S. Chawla. 2020. ML-based Cross-Platform Query Optimization. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1489–1500.

[22] Mark G Kelly, David J Hand, and Niall M Adams. 1999. The Impact of Changing Populations on Classifier Performance. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. 367–371.

[23] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *arXiv preprint arXiv:1809.00677* (2018).

[24] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating Cardinalities with Deep Sketches. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1937–1940. https://doi.org/10.1145/3299869.3320218

[25] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018). arXiv:1808.03196

[26] Sebastian Kruse, Zoi Kaoudi, Bertty Contreras-Rojas, Sanjay Chawla, Felix Naumann, and Jorge-Arnulfo Quiané-Ruiz. 2020. RHEEMix in the Data Jungle: a Cost-based Optimizer for Cross-platform Systems. *VLDB JOURNAL* (2020). https://doi.org/10.1007/s00778-020-00612-x

[27] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[28] Charles X. Ling and Jun Du. 2008. Active Learning with Direct Query Construction. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Las Vegas, Nevada, USA) *(KDD '08)*. Association for Computing Machinery, New York, NY, USA, 480–487. https://doi.org/10.1145/1401890.1401950

[29] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active Learning for ML Enhanced Database Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 175–191. https://doi.org/10.1145/3318464.3389768

[30] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[31] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Houston, TX, USA) *(aiDM'18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 4 pages. https://doi.org/10.1145/3211954.3211957

[32] Ryan Marcus and Olga Papaemmanouil. 2019. Flexible Operator Embeddings via Deep Learning. *CoRR* abs/1901.09090 (2019). arXiv:1901.09090 http://arxiv.org/abs/1901.09090

[33] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (July 2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[34] Volker Markl and Guy M. Lohman. 2002. Learning Table Access Cardinalities with LEO. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki (Eds.). ACM, 613. https://doi.org/10.1145/564691.564766

[35] Nicolai Meinshausen. 2006. Quantile Regression Forests. *Journal of Machine Learning Research* 7, Jun (2006), 983–999.

[36] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1049–1058.

[37] Meikel Poess and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Rec.* 29, 4 (Dec. 2000), 64–71. https://doi.org/10.1145/369275.369291

[38] Adrian Daniel Popescu, Andrey Balmin, Vuk Ercegovac, and Anastasia Ailamaki. 2013. PREDIcT: Towards Predicting the Runtime of Large Scale Iterative Analytics. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1678–1689. https://doi.org/10.14778/2556549.2556553

[39] Alexander Ratner, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proc. VLDB Endow.* 11, 3 (Nov. 2017), 269–282. https://doi.org/10.14778/3157794.3157797

[40] Alexander Ratner, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak

Supervision. *Proc. VLDB Endow.* 11, 3 (Nov. 2017), 269–282. https://doi.org/10.14778/3157794.3157797

[41] Lior Rokach and Oded Maimon. 2005. *Clustering Methods*. Springer US, Boston, MA, 321–352. https://doi.org/10.1007/0-387-25465-X_15

[42] Justin Salamon and Juan Pablo Bello. 2017. Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification. *IEEE Signal Processing Letters* 24, 3 (2017), 279–283.

[43] Erwan Scornet. 2016. On the asymptotics of random forests. *Journal of Multivariate Analysis* 146 (2016), 72 – 83. https://doi.org/10.1016/j.jmva.2015.06.009 Special Issue on Statistical Models and Methods for High or Infinite Dimensional Spaces.

[44] Burr Settles. 2009. *Active Learning Literature Survey*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[45] Connor Shorten and Taghi M Khoshgoftaar. 2019. A Survey on Image Data Augmentation for Deep Learning. *Journal of Big Data* 6, 1 (2019), 60. https://doi.org/10.1186/s40537-019-0197-0

[46] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 19–28.

[47] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-Based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 307–319. https://doi.org/10.14778/3368289.3368296

[48] Balder ten Cate, Phokion G. Kolaitis, Kun Qian, and Wang-Chiew Tan. 2018. Active Learning of GAV Schema Mappings. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Houston, TX, USA) *(SIGMOD/PODS '18)*. Association for Computing Machinery, New York, NY, USA, 355–368. https://doi.org/10.1145/3196959.3196974

[49] Jonas Traub, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2019. Agora: Bringing Together Datasets, Algorithms, Models and More in a Unified Ecosystem [Vision]. *Proc. VLDB Endow.* 49, 4 (Dec. 2019), SIGMOD Record.

[50] Yiwei Wang, Wei Wang, Yuxuan Liang, Yujun Cai, Juncheng Liu, and Bryan Hooi. 2020. NodeAug: Semi-Supervised Node Classification with Data Augmentation. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 207–217. https://doi.org/10.1145/3394486.3403063

[51] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal Component Analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.

[52] Wentao Wu, Xi Wu, Hakan Hacigümüş, and Jeffrey F. Naughton. 2014. Uncertainty Aware Query Execution Time Prediction. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 1857–1868. https://doi.org/10.14778/2733085.2733092

[53] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 279–292. https://doi.org/10.14778/3368289.3368294

[54] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016), 56–65.

[55] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries Using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (May 2020), 1416–1428. https://doi.org/10.14778/3397230.3397238

[56] Xiangxin Zhu, Carl Vondrick, Charless C Fowlkes, and Deva Ramanan. 2016. Do we need more Training Data? *International Journal of Computer Vision* 119, 1 (2016), 76–92.