Farming Your ML-based Query Optimizer's Food

Robin van de Water[3*] Francesco Ventura[1] Zoi Kaoudi[1]
Jorge-Arnulfo Quiané-Ruiz[1,2] Volker Markl[1,2]

[1]Technische Universität Berlin (TU Berlin) [2] DFKI GmbH [3]Hasso Plattner Institut (HPI)
robin.vandewater@hpi.de {francesco.ventura, zoi.kaoudi, jorge.quiane, volker.markl}@tu-berlin.de

***Abstract*—Machine learning (ML) is becoming a core component in query optimizers, e.g., to estimate costs or cardinalities. This means large heterogeneous sets of labeled query plans or jobs (i.e., plans with their runtime or cardinality output) are needed. However, collecting such a training dataset is a very tedious and time-consuming task: It requires both developing numerous jobs and executing them to acquire ground-truth labels. We demonstrate DATAFARM, a novel framework for efficiently generating and labeling training data for ML-based query optimizers to overcome these issues. DATAFARM enables generating training data tailored to users' needs by learning from their existing workload patterns, input data, and computational resources. It uses an active learning approach to determine a subset of jobs to be executed and encloses the human into the loop, resulting in higher quality data. The graphical user interface of DATAFARM allows users to get informative details of the generated jobs and guides them through the generation process step-by-step. We show how users can intervene and provide feedback to the system in an iterative fashion. As an output, users can download both the generated jobs to use as a benchmark and the training data (jobs with their labels).**

I. INTRODUCTION

Machine learning (ML) has gained a prominent role in query optimization both in academia and industry. Most of the proposed techniques are based on supervised learning and thus require the acquisition of valuable data to train the models on. The effectiveness of such models depends on the quantity and quality of data and the availability of labels. This requirement of large amounts of high-quality data quickly becomes a roadblock in the context of query optimization. First, collecting many real query plans (jobs) with labels (e.g., execution time or cardinality values) requires developing thousands of heterogeneous plans, both optimal and sub-optimal. Second, after gathering these thousands of plans, one must execute them to get their label. The latter is very time-consuming, as it leads to the execution of numerous jobs, including sub-optimal ones. For example, collecting labels for only 500 OLAP Flink jobs with input data of about 1TB in our four-quadcore-nodes cluster takes almost 10 days. This duration is problematic because learning a model typically requires several thousands of jobs. Extrapolating this to $10,000$ jobs would require more than 6 months! Even if logs are available, the query plans contained in the logs are the ones the optimizer chose to execute, and thus most of them are (near-)optimal. Using only these plans would lead to a biased ML model, which would be ignorant of the performance of "bad" plans.

Surprisingly, there has been little progress in tackling the problem of generating labeled query workloads. Most works on ML-based optimizers still assume the availability of training data exploiting both private and public, often synthetic, datasets to develop and test their solutions [2], [4], [5], [11]. In contrast to other domains, where advanced data augmentation techniques play a significant role [6], just a few preliminary attempts have been made in data management [1]. We still rely on task-specific benchmarking workload generators, such as TPC-H or TPC-DS, which produce homogeneous workloads with low variance from a few fixed patterns. Furthermore, none of these benchmarks provide labeling: one still has to execute a vast amount of queries.

To tackle this challenge, we recently proposed DATAFARM in the context of Agora [8]. DATAFARM is a novel framework for generating training data for learned query optimizers [10]. DATAFARM follows a data-driven white-box approach to augment an initial small query workload and attach labels (e.g., runtime) to each generated query. Its benefits are numerous: (i) Its data-driven approach allows for augmenting an initial query workload so that the newly generated plans fit users' needs; (ii) Its execution strategy allows for selectively running generated plans and imputing label estimates to the non-executed plans, which reduces label collection time drastically, and; (iii) Its white-box strategy allows users to understand and debug every step of the process and trust its outcome.

Our demo will showcase the three main components of DATAFARM [10] and its newly added feature that encompasses the human in the training data generation [9]. The main idea behind the new feature is that users know their query workload better and can thus interact with the system to ultimately obtain training data efficiently and with high quality. To achieve this, users can analyze all the jobs via an intuitive and easy-to-use graphical user interface (GUI) that provides several insights into the generation process. In summary, we will demonstrate how DATAFARM: (i) guides users through the entire training data generation process step-by-step via its GUI; (ii) walks users through its data visualizations to better understand and compare the query generated workloads with the initial query workload; (iii) allows users to intervene in the generation process to provide feedback iteratively; (iv) enable users to not only get training data but also download the generated queries to be used as a query workload benchmark.

*Work done while conducting his master thesis at TU Berlin.

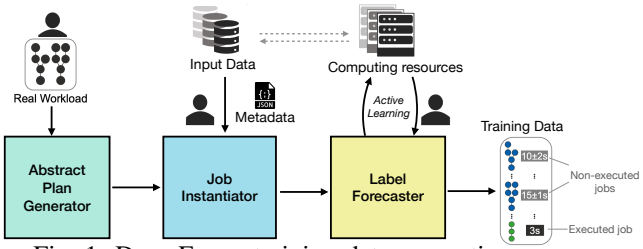

Fig. 1: DATAFARM training data generation process.

II. DATAFARM OVERVIEW

Our goal is to produce a large volume of training data (i.e., query workload with labels) for ML-based query optimizers in a reasonable amount of time. We aim for a data-driven white-box augmentation strategy that additionally provides labels for each query. Developing an efficient and reliable training data generation framework comes with many challenges: (i) Jobs should be representative of the existing small workload and, at the same time, heterogeneous; (ii) Jobs should be executable and realistic; (iii) We have to generate reliable labels without executing all generated jobs; (iv) Considering that we afford to execute a small set of queries only, we need to determine which is the smallest set of representative queries to execute; (v) We have to determine the best information and visualization that aids the human towards queries that should be executed.

We tackle all these challenges with DATAFARM. The input of DATAFARM is a (typically small) set of dataflow jobs, and the output is an augmented (much larger) set of jobs together with a label (e.g., runtime or output cardinality). DATAFARM comprises three main components [10]: (i) the *Abstract Plan Generator*, (ii) the *Synthetic Job Instantiator*, and (iii) *Label Forecaster* (Figure 1). We briefly explain each component and detail how the human aids in the labeling phase.

A. Abstract Plan Generator

The goal of the *Abstract Plan Generator* (APG) is to learn the patterns that appear in the input workload to generate new jobs that both contain meaningful sequences of operators and are representative of the real ones (*challenges (i)&(ii)*). APG takes as input a real workload of jobs, learns the relations and patterns between the operators, and generates various new heterogeneous *abstract plans*. An abstract plan is a DAG of operators without an actual operator implementation, such as selection predicates for `Filter` operators or join keys for `Join` operators. In more detail, APG proceeds in two phases: the *query patterns learning* and the *query plans generation*.

(1) *Query Patterns Learning*. APG first learns the patterns of the input jobs by analyzing each input plan as a Markov Chain. Each operator in the plan represents a possible state of the system independent from the previous and the next one. APG learns two transition matrices (parent and child) which, given one operator, contain the probability of each possible transition up and down in the plan. It computes these probabilities based on the relative frequency of an operator in each input plan.

(2) *Query Plans Generation*. Then, given the parent and child transition matrices, APG creates abstract plans. To do so, it iteratively samples the probability mass function of the current operator from the child transition matrix and decides which should be the next operator to insert in the abstract plan. Whenever the algorithm encounters a `Join` operator, it consults the parent matrix to build the plan backward. Note that probabilities are used as weights when sampling and, thus, a transition having zero probability will never appear in any abstract plan. These weights ensure that our generated abstract plans are always valid and realistic.

B. Synthetic Job Instantiator

The abstract plans outputted by APG cannot be executed yet for two reasons. First, they are not concrete jobs and, second, the operators' user-defined functions (UDFs), such as selection predicates, have not been instantiated yet. The goal of the *Synthetic Job Instantiator* (SJI) is to generate a representative and heterogeneous query workload that can be successfully executed (*challenges (i)&(ii)*). It is thus of primary importance to instantiate UDFs tailored to the actual data distribution. Managing the wide range of UDFs along with their parameters and the possible join orderings that can be implemented is quite challenging. SJI takes as input the abstract plans generated by APG. For each of them, it creates different *job instances*. It combines possible input parameters extracted from the input data metadata, such as the distribution of filterable values and the input data schema. This metadata primarily comprises structural information and statistics that add significant value to the instantiation process. 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). We note that SJI leverages statistical analysis of the input data and exploits the input metadata to ensure a realistic instantiation for each operator, ensuring that they are always executable. For example, `Joins` are performed only among joinable fields. DATAFARM provides a handful of interfaces allowing users to extend the default operators' instantiation with custom user's UDFs [10].

C. Active Learning Label Forecaster

Executing a large query workload to get the labels is impractical as discussed earlier. DATAFARM overcomes this limitation by determining a small subset of jobs to be executed while forecasting the labels of the non-executed jobs (*challenges (iii)&(iv)*). The *Label Forecaster* (LF) exploits an active learning approach and uses quantile regression forests (QRF). The principle of LF is to execute only the most informative jobs and forecasting the non-executed jobs' labels using a QRF model. Thus, iteration after iteration, LF executes only jobs that add new information to the QRF model.

It is evident that the human alone cannot manually select among thousands of jobs that must be executed. For this reason, at each iteration of active learning, LF suggests a subset of jobs by using PCA to extract a small set of principal

components from the jobs' features. Then, it identifies a user-specified number of groups of jobs through agglomerative clustering and selects the centroids of the clusters as a sample of jobs to propose to the user. This procedure allows one to execute jobs with maximum inter-differences. Subsequently, users can refine this subset to one that they believe will lead to a better QRF model and, thus, a better quality of the training data. Next, LF trains a predictive model with the features of the executed jobs and the collected runtimes (or output cardinality). Finally, it exploits the features of the non-executed jobs and the model to predict the missing labels and attach uncertainty values to them. The uncertainty forms one of the aspects that helps the user to select the jobs to execute in the subsequent iterations more easily (*challenge (v)*).

Every new iteration is potentially very costly due to the jobs' execution time; we, therefore, use a stopping condition to meet a good trade-off between predictive performance and the number of executed jobs. DATAFARM takes a conservative approach on the number of executed jobs and suggests stopping every time the model's uncertainty shows a significant drop. The user then decides whether to stop or continue the labeling process. The output of the whole process is an augmented dataset of labeled jobs tailored to the input query workload, input data, and computational resources.

D. Results

To evaluate the quality of our training data, we generated 2000 jobs from only 6 TPC-H queries. We construct four training sets that differ on the process of obtaining the labels: The first training set contains ground truth labels that we obtain after executing all jobs, while the other three contain labels acquired by sampling 166 jobs to execute and using the ML model of the Label Forecaster to predict the labels of the rest. We used three different sampling mechanisms: (i) random, (ii) agglomerative clustering (DATAFARM *without-the-human*), and, (iii) manually modifying the set suggested by the agglomerative clustering (DATAFARM *with-the-human*). Based on these four training sets, we build four QRF models, respectively, and predict our input query workload runtimes using each model.

Figure 2 shows the root mean square error of each model. We observe that the quality of training data generated by DATAFARM is as good as the ground truth (green and blue bars, respectively). Most notably, we observe that DATAFARM (with-the-human) outperforms not only DATAFARM (without-the-human) but also the ground truth model. This result is possible because the user can determine the essential features for the ML model and make modifications according to her observations (adding and removing jobs). Note that DATAFARM *with-the-human* can achieve better performance than when obtaining labels by executing the complete set of jobs because overfitting can be avoided. We observed that the combination of job selection time and execution time is similar for the three sampling mechanisms described above. Thus, DATAFARM *with-the-human* does not significantly increase the total sampling duration, despite manual involvement. We observed similar results, in terms of prediction performance and sampling duration, for three other query workloads composed of over 3000 jobs, based on real-world IMDB data.

Fig. 2: Models' RMSE, trained on sets with different labels.

III. DEMONSTRATION

The goal of our demonstration is to enable the audience to understand: (i) how the different phases of DATAFARM are crucial for generating training data for ML-based optimizers; (ii) how the human-in-the-loop can help this process, and (iii) how one can use DATAFARM to combine any input query workload with any input dataset to generate queries and training data. We will demonstrate DATAFARM using the real-world IMDB dataset to generate Apache Flink[1] jobs.

A. Scenarios

We assume two users: *Bob* and *Alice*. Bob is a system administrator responsible for the ML-based optimizer of System X. The optimizer includes an ML model that predicts the runtime of each query plan, which is then used as a cost during the plan enumeration. Thus, Bob needs training data (i.e., queries with their runtime as labels) to build his ML model. On the other side, Alice is an administrator of a traditional cost-based System Y. Although she does not require training data, she still needs a large query workload to benchmark, profile, and tune System Y.

The audience will be able to play the role of either Alice or Bob, which consists of three main steps: (1) generating abstract plans, (2) instantiating the abstract plans for execution, and (3) labeling the plans. Steps (1) and (2) are similar for both Alice and Bob. Step (3) targets users like Bob.

Step (1) – *Abstract Plan Generation.* Alice (or Bob) uploads a small query workload. DATAFARM analyzes it and provides various visualizations and statistics: e.g., it clusters the jobs based on a similarity metric that Alice chooses (e.g., the number of operators). Alice can also inspect and modify the children and parent matrices created by the Abstract Plan Generator. She then chooses the number of abstract plans to generate, the maximum number of operators per plan, and the total number of joins per plan. Once DATAFARM generates the abstract plans, Alice can inspect the plans via a scrolling window and go back to modify some initial knobs and generate new abstract plans from scratch if necessary.

Step (2) – *Jobs Instantiation.* As the abstract plans generated in step (1) cannot be executed yet, the next step is to instantiate them, i.e., to create the actual jobs. By default, DATAFARM instantiates all plans. However, assume that Alice does not want plans with iterations, so she removes them from the job's instantiation process. She also specifies the number of instances per plan and configures the generated queries to

[1] https://flink.apache.org/

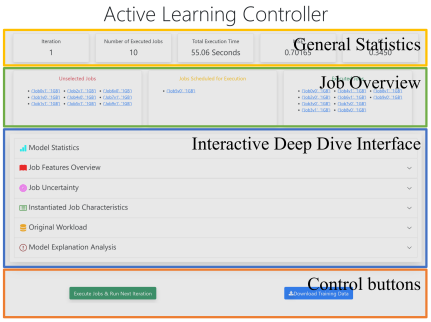

Fig. 3: Active Learning Controller

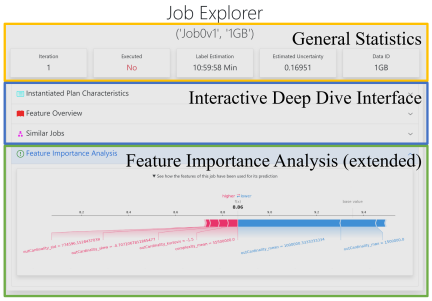

Fig. 4: Job Explorer

be executed as Flink jobs. DATAFARM generates new jobs and returns statistics and visualizations about the generated workload. If she is not satisfied with the new jobs, she can still go back to any step of the generation process. At this point, she downloads the generated jobs and uses them to tune System Y. Bob, in contrast, needs the execution time of the generated jobs to use in his ML model training. He, thus, decides to proceed with the labeling process provided by DATAFARM.

Step (3) – *Labels Generation.* Bob is the main actor in the active learning process. He specifies the initial number of jobs that have to be executed, provides an uncertainty threshold, a maximum number of iterations, and his time budget; he also determines whether DATAFARM should proceed with executing the suggested candidate set of jobs or a modified set. He does so by inspecting the various facets of the active learning controller shown in Figure 3 and selecting or deselecting jobs for execution. DATAFARM then runs these jobs and provides Bob with a visualization of all jobs and their uncertainty levels. Bob can then zoom in to observe specific jobs or check their feature importance via the Job Explorer (see Figure 4). If he is happy with the results, he downloads the training data and starts building the ML model for his query optimizer. If the results are not satisfactory, Bob can instruct DATAFARM to execute more jobs so that the model accuracy improves.

B. User Experience

DATAFARM comes with a user-friendly interactive GUI that will smoothly walk the audience through the training generation process. Most notably, the GUI facilitates users to explore and modify the suggested set of candidate jobs, thus improving user experience. The audience will be able to browse the suggested jobs, add or remove jobs, execute the next iteration, and download the training data with the current forecasted labels. The GUI consists of two main parts: the *Active Learning Controller* and the *Job Explorer*.

The Active Learning Controller (Figure 3) includes main information statistics, such as the number of executed jobs and the model performance so far. Furthermore, it has several collapsible sections, in accordance with the progressive disclosure principle [7]: performance statistics, job features, uncertainty values of individual jobs, instantiated job characteristics, and model explainability. For instance, users can identify the current model performance and determine if they will continue to run extra iterations of the label forecaster to increase accuracy. Another example of helpful information includes the model explainability analysis, which consists of several model explanation visualizations generated using the SHAP [3] library. The audience will be able to realize how much each feature contributes to model predictions and decide to execute particular jobs to gain a better model.

The Job Explorer (Figure 4) follows the same principle and displays detailed information about one particular job. For example, it contains a "similar jobs" section, which shows executed jobs comparable in operator length, cardinalities, and data sources. This interface element helps a user with the job selection process. For instance, if the label of a particular job deviates significantly from similar jobs, the user can decide to execute this job to increase model performance.

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] Z. Kaoudi, J. Quiané-Ruiz, B. Contreras-Rojas, R. Pardo-Meza, A. Troudi, and S. Chawla. ML-based cross-platform query optimization. In *ICDE*, pages 1489–1500, 2020.

[2] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.

[3] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In *NIPS*, pages 4765–4774. 2017.

[4] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, July 2019.

[5] R. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.*, 12(11):1733–1746, 2019.

[6] A. Ratner, S. H. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré. Snorkel: Rapid training data creation with weak supervision. *Proc. VLDB Endow.*, 11(3):269–282, Nov. 2017.

[7] A. Springer and S. Whittaker. Progressive Disclosure: Empirically Motivated Approaches to Designing Effective Transparency. In *IUI*, page 107–120, 2019.

[8] J. Traub, Z. Kaoudi, J.-A. Quiané-Ruiz, and V. Markl. Agora: Bringing together datasets, algorithms, models and more in a unified ecosystem [vision]. *SIGMOD Record*, 49(4), 2021.

[9] R. van de Water, F. Ventura, Z. Kaoudi, J. Quiané-Ruiz, and V. Markl. Farm Your ML-based Query Optimizer's Food! – Human-Guided Training Data Generation –. In *CIDR*, 2022.

[10] F. Ventura, Z. Kaoudi, J. Quiané-Ruiz, and V. Markl. Expand your Training Limits! Generating and Labeling Jobs for ML-based Data Management. In *SIGMOD*, pages 1865–1878, 2021.

[11] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.*, 13(3):279–292, Nov. 2019.