PerfGuard: Deploying ML-for-Systems without Performance Regressions

H M Sajjad Hossain, Lucas Rosenblatt, Gilbert Antonius, Irene Shaffer, Remmelt Ammerlaan, Abhishek Roy, Markus Weimer, Hiren Patel, Marc Friedman, Shi Qiao, Peter Orenberg, Soundarajan Srinivasan, Vijay Ramani, Alekh Jindal
perfguard@microsoft.com

ABSTRACT

There is newer trend of applying machine learning to systems (ML-for-Systems), i.e., leveraging the large workloads that are available in modern cloud applications for improving the system performance. However, learning over cloud workloads commonly leads to over generalizations that do not capture the large variety of workload patterns. As a result, ML-for-System approaches tend to augment performance for some subset of the workload, while risking severe performance regressions in other subsets. In this paper, we describe the vision of a performance safeguard system, system, that helps design pre-production experiments for determining the production readiness of learned models. Concretely, we consider the big data processing infrastructure at Microsoft and consider deploying a large set of learned cardinality models for its query optimizer. We describe an experimentation pipeline that differentiates the impact of query plans with and without the learned cardinality models, selects plan differences that are likely to lead to most dramatic performance difference, runs a constrained set of pre-production experiments to empirically observe the runtime performance, and finally picks the models that are expected to lead to consistently improved performance for deployment. PerfGuard enables safe deployment not just for learned cardinality models but also for a plethora of other ML-for-Systems features.

1 INTRODUCTION

System features often require empirical evidence before they could be deployed to production. This is especially a pain in modern cloud services with faster release cycles and bug fix iterations. The typical practice is to run a subset of production workloads, typically chosen manually using best effort, in a pre-production environment and determine the production readiness from the results. Designing these pre-production experiments is already a challenge, however, the problem gets worse with the newer trend of applying ML to systems, i.e., features that incorporate machine learning models to improve system behavior. These ML-based features can consist of a large set of models that are often complex and hard to reason about. As a result, it is difficult to manually design the experiments that identify the safe-to-deploy models, i.e., ones which consistently lead to improved system behavior in production. Therefore, we need to automate the way we design systems experiments for testing the newer breed of ML-for-Systems features.

Specifically, consider the cardinality estimation problem. Cardinality is a key statistic used by query optimizers to pick the physical query plans. However, the accuracy of the current cardinality estimators is often way off. For instance, the cardinality estimation in SCOPE query engine [3] could range anywhere from 10,000 times under-estimation to 1 million times over-estimation, resulting in query plans that are suboptimal both in performance and in resource consumption. The problem is ubiquitous in all modern query engines, such as Spark, SQL Server, PostgreSQL, MySQL, etc. As a result, learning cardinality models has recently gained a lot of attention in both academia and industry [7][2][5][4]. Our recent work, CardLearner [6], exploits patterns in SCOPE workloads to learn multiple small models that are several orders of magnitude more accurate at predicting cardinalities. To deploy the above cardinality models to production, we need to test their runtime behavior in a pre-production environment with constraint on the number of jobs we can re-run. In this paper, we test our pipeline to select and rerun a subset of jobs which validate the accuracy of the cardinality models and safeguard the performance of SCOPE query engine.

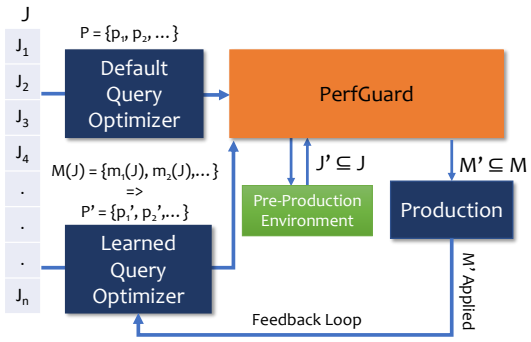

Figure 1: An instance of performance safeguarding problem for deploying a set of learned cardinality models M over a query workload J seen by a query optimizer.

2 PROBLEM FORMULATION

Let us consider a set of machine learning models, $M = \{m_1, m_2, .., m_p\}$ to be deployed on a set of jobs, $J = \{j_1, j_2, \ldots, j_n\}$. In the process, we maintain a set of binary variables $A = a_{11}, a_{12}, \ldots, a_{ik}$ which indicate whether i^{th} model is applicable to k^{th} job or not. However, models learned from previous jobs may lead to performance regression over future jobs. Therefore, given a budget B for the number of jobs we can execute in the pre-production setup, we need to select a set of candidate jobs to run. The goal is to filter out the models that cause regression and flag the remaining set of models M' for production. Figure 1 summarizes this performance safeguarding problem when deploying learned cardinality models for a query optimizer. However, note that that the problem is applicable in general to the wider set of cloud systems that aim to continually learn from their past workloads.

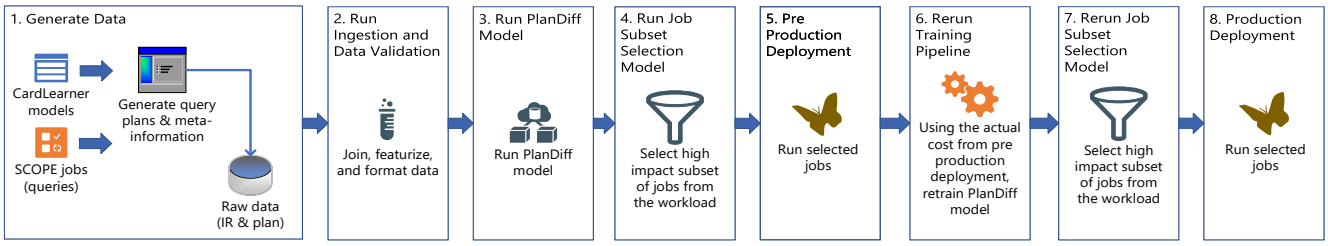

Figure 2: Overall view of PerfGuard pipeline.

3 METHODOLOGY

Database query engines compile user queries into an optimized physical query plan q that can be represented as a Directed Acyclic Graph (DAG). We leverage these DAG representations of query plans to learn and isolate regression causing characteristics as follows. For a job j_i, we generate a default query plan q_i. We then apply a model subset $m \subset M$ to produce a second query plan, q_i'. Using our experimentation pipeline, we attempt to distinguish these two plans before execution by applying $D(q_i, q_i') = \theta$. D can be any arbitrary model that consumes q and q', producing an estimation of θ (we chose *graph convolution* as D for experimentation). Domain expertise informs the choice of feedback signal θ, which should reflect an impactful metric in differentiating query plans (for θ, we chose *cost of execution* when experimenting).

Figure 2 illustrates the steps in PerfGuard. Step 1 takes the CardLearner models and SCOPE jobs and runs a data generation script to produce the the pre- and post-CardLearner physical plans along with their corresponding meta-information. We then combine the raw data from the two sources merging the node level features into an intermediate representation (IR) and keeping the physical plans as graph structures. Step 2 runs the ingestion and data validation pipeline to turn the data into the proper featurized Perfguard model format. Step 3 runs the PlanDiff model. Step 4 is the job subset selection module to pick a subset of high-impact jobs from the workload and featurizes the selected job subset meta data. Step 5 starts pre-production deployment and runs the selected jobs to obtain run time information. Step 6 is the training pipeline that re-runs using the actual cost from pre-production deployment in order to improve accuracy. Step 7 begins the CardLearner selection for the production environment and provides a list of CardLearner models that are safe to run for each job. Finally, production deployment begins in Step 8.

4 EXPERIMENTAL EVALUATION

In order to safeguard the performance of CardLearner models using our pipeline, we attempt to predict the normalized cost difference between the original plan generated by the default query engine and the modified plan using the estimates of CardLearner models. We compare the performance of our model with baseline regression models. For these baseline models, we leverage only the IR statistics of both the query plans. For a job j_i, ${I_1}^i$ and ${I_2}^i$ are the IR matrices for query plans ${q_1}^i$ and ${q_2}^i$. In order to formulate a combined feature representation we multiply both IR matrices and then extract histogram features with a fixed bin size of 20. As a result each pair of query plans will be represented as feature according to, $f_i = \sum_{j=1}^{20} hist({I_1}^i \times {I_2}^i)$. We employed two regression models based on XgBoost and a two layer feed forward neural network.

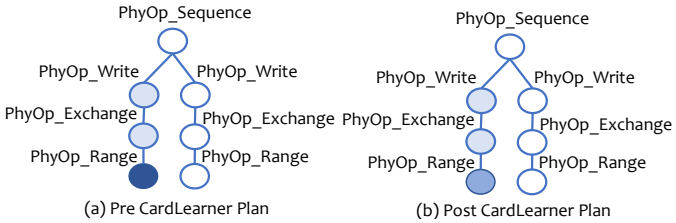

Figure 3: Visualizing importances of physical operators.

We trained both algorithms using 3000 job pairs and tested with 100 job pairs. We demonstrate the performance of our baseline regression models in Table 1 with different evaluation metrics. We see that XgBoost outperforms DNN and our DNN is performing really bad in terms of explaining the variance according to the R^2 score. However low R^2 does not always indicate that it is a bad model.

	MSE	MAE	R^2 Score
XgBoost	0.09	0.22	0.19
DNN	0.13	0.27	0
PerfGuard	0.10	0.22	0.08

Table 1: Performance of different approaches

In our pipeline, we leverage both query graph structure and IR data and apply graph convolution network to learn node embeddings and then aggregate the embeddings to formulate graph embeddings for both query graph structures. We adopt the architecture proposed in SimGNN [1] to calculate the similarity between the two plans. We utilize attention mechanism to aggregate the node embeddings, which also provides the importance scores of the physical operators in the query plans. In Figure 3 we plot the pre and post CardLearner query plan for a job. According to our algorithm, the similarity score between these plans is - 0.63702 which means the plans are almost similar. However if we look at the list of physical operators in each plans, we notice that there is no change in the plan. We further investigated individual IRs and validated that the IRs are actually different for the highlighted nodes which made some differences between the two plans. We report the accuracy of our algorithm in Table 1. Although our model performance exhibits similar performance to XgBoost, but our algorithm provides a method to interpret the difference.

5 CONCLUSION

Safely deploying the newer breed of ML-for-Systems features is a challenge in cloud systems. In this paper, we presented PerfGuard, a vision for safeguarding performance regression in production. Initial results with learned cardinality models demonstrate at least on-par performance with baseline models. Further investigation of model architecture and larger datasets should improve performance significantly.

REFERENCES

[1] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. SimGNN: A Neural Network Approach to Fast Graph Similarity Computation. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining, WSDM 2019, Melbourne, VIC, Australia, February 11-15, 2019*. 384–392.

[2] Rajesh Bordawekar and Oded Shmueli (Eds.). 2019. *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*. ACM. https://doi.org/10.1145/3329859

[3] Ronnie Chaiken, Bob Jenkins, Per-undefinedke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1265–1276.

[4] Hazar Harmouch and Felix Naumann. 2017. Cardinality Estimation: An Experimental Survey. *PVLDB* 11, 4 (2017), 499–512.

[5] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2019. An Empirical Analysis of Deep Learning for Cardinality Estimation. *CoRR* abs/1905.06425 (2019). arXiv:1905.06425

[6] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *PVLDB* 12, 3 (2018), 210–222.

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