PerfGuard: Deploying ML-for-Systems without Performance Regressions, Almost!

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

ABSTRACT

Modern data processing systems require optimization at massive scale, and using machine learning to optimize these systems (ML-for-systems) has shown promising results. Unfortunately, ML-for-systems is subject to over generalizations that do not capture the large variety of workload patterns, and tend to augment the performance of certain subsets in the workload while regressing performance for others. In this paper, we introduce a performance safeguard system, called *PerfGuard*, that designs pre-production experiments for deploying ML-for-systems. Instead of searching the entire space of query plans (a well-known, intractable problem), we focus on query plan deltas (a significantly smaller space). PerfGuard formalizes these differences, and correlates plan deltas to important feedback signals, like execution cost. We describe the deep learning architecture and the end-to-end pipeline in PerfGuard that could be used with general relational databases. We show that this architecture improves on baseline models, and that our pipeline identifies key query plan components as major contributors to plan disparity. Offline experimentation shows PerfGuard as a promising approach, with many opportunities for future improvement.

PVLDB Reference Format:
Remmelt Ammerlaan, Gilbert Antonius, Marc Friedman, H M Sajjad Hossain, Alekh Jindal, Peter Orenberg, Hiren Patel, Shi Qiao, Vijay Ramani, Lucas Rosenblatt, Abhishek Roy, Irene Shaffer, Soundarajan Srinivasan, Markus Weimer . PerfGuard: Deploying ML-for-Systems without Performance Regressions, Almost!. PVLDB, 14(13): 3362 - 3375, 2021.
doi:10.14778/3484224.3484233

1 INTRODUCTION

ML-for-systems, the new breed of features that apply machine learning to improve system behavior, is fast emerging as a design principle for modern data processing systems. These systems have become incredibly complex [42], making learning-based methods an attractive approach for automatic optimization over different instances of a workload [22, 42]. ML-for-systems is further facilitated by the presence of large volumes of workload telemetry that can be used for training in modern cloud deployments [17]. Consequently, a number of learning-based features have recently been proposed to optimize various parts of a query processing system, including, data structures like indexes [8, 20, 23, 29] and bloom filters [23, 40], query estimates such as cardinality [9, 10, 12–15, 19, 31, 34, 41, 42, 45, 46, 48] and costs [1, 11, 25, 37, 39, 43], query planners [27, 30], schedulers [26], or even the entire query optimizers [28]. These features replace portions of the query processing system that are difficult to tune manually with models learned over past workloads, thus helping to adapt to newer cluster conditions.

Unfortunately, deploying ML-for-systems is a major challenge. In fact, even highly accurate ML models, when integrated with the end-to-end query processing systems, can augment performance for some subset of the workload, while risking severe performance regressions in other portions of the workload. For example, on SCOPE [4], the big data query system at Microsoft, we observed performance regressions with learned cardinality models [42], learned cost models [37], learned degree of parallelism [36], and even when steering the query planner [30]. There are several reasons for these regressions. First, a behavior learned from past workloads risks overgeneralizing as the cost function associated with learning seeks to maximize average performance and may not apply to the large variety of workload patterns in cloud environments. Second, introducing learned models leads to novel query execution plans with unexpected performance metrics. And finally, query processing systems comprise of several disjoint moving parts, like cardinality and cost estimation, which are ten fed into the query planner. While it is practical to learn models for each of the components independently and not end up with a giant "black box" solution, the individual models often interact in unseen and unpredictable ways. A learned cost model may behave differently when fed with learned cardinalities rather than traditional cardinality estimates. Thus, successfully deploying learning-based features to production requires solving a meta-optimization task: *minimize the overall expected performance regressions introduced by specific optimizations.*

The current practice in Cosmos is to run pre-production validation before each SCOPE optimizer release. It involves manually sampling a subset of production workload, based on business importance or workload diversity, for re-execution (*flighting*) in a pre-production environment. Flighting results in performance metrics with both the old and new SCOPE runtime, which the SCOPE engineers compare, paying particular attention to performance regressions which must be investigated further before the final sign off. The above approach is both resource and labor intensive: tens of thousands of jobs need to be flighted (incurring large resource

Proceedings of the VLDB Endowment, Vol. 14, No. 13 ISSN 2150-8097.
doi:10.14778/3484224.3484233

cost) and hundreds of performance regressions are manually investigated (eating up precious developer time). Furthermore, the sampling process is not representative of the overall workload. The other approach is to enable new SCOPE features in a tier-wise manner. However, this requires explicit acknowledgement from key customers in each tier, who in turn want to know the regressions to expect and the plan to mitigate. Likewise, opting out jobs in a post-hoc manner can pile the support load with too many customer incident reports. Therefore, both the tier-wise and the post-hoc approaches can only be applied when performance impact is well understood, something which is not the case for learned models.

Recent work has studied the impact of query plan changes in the context of automated indexing [7]. It leverages run-time statistics from past queries that used indexes and trains a partially connected HybridDNN to learn the performance impact. In contrast, we have a cold start problem where the ML-based optimizations are not enabled in the first place and so we don't have the run-time statistics. Rather, we need to design the pre-production experiments to build confidence to deploy the ML-based optimizations.

In this paper, we present a systematic approach to avoiding performance regressions when deploying ML-based features. Such an approach will catch and prevent performance regression for releasing ML-based query optimizer features to specific customers and workloads. Given that pre-production resources are limited, our goal is to carefully select the jobs to flight and build confidence in deploying learned models, like new versions of the query processing system. Our key intuition is that performance regressions can be predicted using the *changes* in physical query execution plans that are caused by learned models. We train such a meta-model by leveraging past flighting data from the pre-production clusters and use this meta-model to predict which jobs are going to regress. We exclude all such jobs from flighting and produce a much smaller portion of the workload to validate and deploy the learned feature on. In summary, we make the following key contributions:

(1) We formalize the performance regression problem with learned features in the context of SCOPE big data processing at Microsoft, and discuss how learning from query plan differences makes this problem tractable by reducing the overall search space. (Section 2)

(2) We introduce PerfGuard, a system for guarding against performance regressions with learning-based optimizations. PerfGuard discovers query plan differences from different versions of a query processing system, trains a machine learning model to predict the performance impact of these differences, and uses these models to flight only a small subset of relevant jobs that are likely to be benefited by the learned features. (Section 3 - 5)

(3) Finally, we present a detailed experimental evaluation over large production workloads and using learned cardinality as a concrete learning-based optimization. Our results show that PerfGuard can predict the performance for plan differences with an absolute error of 0.12, and help reduce the set of jobs to flight with only ≈ 20% incorrect selection. (Section 6)

2 PERFORMANCE REGRESSION

We focus on the Cosmos big data analytics platform at Microsoft to understand the performance regression problem. Cosmos uses SCOPE [4, 47], a SQL-like engine to run hundreds of thousands of jobs processing petabytes of data per day. The SCOPE query optimizer translates a SCOPE job j_i, into an optimized physical query plan p_i, describing how a query is executed physically in the cluster. The physical query plan is a Directed Acylic Graph (DAG) of nodes, with each node corresponding to physical database operations, such as scan, filter, join, aggregate, etc. Applying optimizations, ML-based or not, to the SCOPE query engine typically affects the physical query plans, and consequently the query performance either improves or regresses. Anticipating the performance impact of these plan changes is not trivial, especially for ML-based optimizations that could consist of complex black box models. As a result, ML-based optimizations make it harder to construct the pre-production experiments for determining their readiness, i.e., it is not obvious which subset of SCOPE workloads should be re-executed in order to build the deployment confidence. We call this the *performance regression problem* and state it formally below.

2.1 Problem Statement

We refer to a set of query scripts, called *jobs*, executed within a time frame as workload J, i.e., $J = \{j_1, j_2, \ldots, j_n\}$, where j_i represents one of n jobs in the workload. Consider a ML-based optimization, *mlopt*, consisting of one or more models: $mlopt = \{m_1, m_2, \ldots, m_k\}$. Example *mlopt* could be fine-grained models for learned cardinalities [42], learned cost models [37], learned query planner hints [30], or learned resource allocation [36]. We assume that the models in *mlopt* have already been validated and are indeed of high quality. However, it is not clear which jobs in J will benefit from *mlopt*; if we know this, we can deploy *mlopt* selectively to jobs that will directly benefit. We assume fine-grained deployment control, where we can decide which analytical query has which ML-based optimizations enabled, something that been made possible in the case of Cosmos with the Peregrine workload optimization platform [17].

Our goal then is to learn a meta-model to predict whether or not a job is going to regress with *mlopt*. We isolate jobs with low prediction confidence and flight them in a pre-production environment. The objective is to minimize the list of these low confidence jobs, which need experimentation, within a given flighting budget B. For simplicity, we consider the unit of flighting budget B to be the number of jobs that could be re-executed in the pre-production environment. Naturally, given that pre-production resources are limited, B is much smaller than the overall workload, i.e., $B \ll |J|$.

A smaller set of flighted jobs not only reduces the experimentation costs, but also reduces the developer time to scrutinize flighting results, making it easier to answer a question like: Which plan changes are detrimental? At the same time, learning the above meta-model is challenging since the space of possible physical query plans in J is very large. Fortunately, in practice, there is a much smaller space of performance differences before and after modification. This leads to a question: *can we learn the performance differences for the (smaller) domain of physical query plans changes*?

2.2 Learning from Differences

We envision to learn from differences, i.e., train a predictor that can identify the impactful *changes* in physical query plans, and associate them with impactful *changes* in query performance. Earlier works showed that differences are often easier to estimate based on

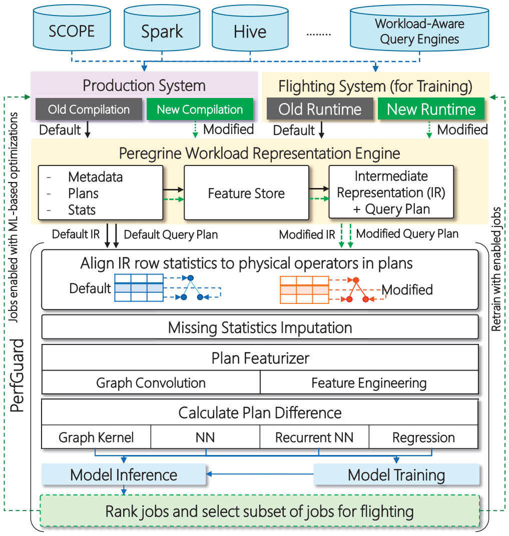

SCOPE
Spark
Hive
....
Workload-Aware Query Engines
Production System
Old Compilation
New Compilation
Default
Modified
Flighting System (for Training)
Old Runtime
New Runtime
Default
Modified
Peregrine Workload Representation Engine
- Metadata
- Plans
- Stats
Feature Store
Intermediate Representation (IR) + Query Plan
Default IR
Default Query Plan
Modified IR
Modified Query Plan
Align IR row statistics to physical operators in plans
Default
Modified
Missing Statistics Imputation
Plan Featurizer
Graph Convolution
Feature Engineering
Calculate Plan Difference
Graph Kernel
NN
Recurrent NN
Regression
Model Inference
Model Training
Rank jobs and select subset of jobs for flighting
Jobs enabled with ML-based optimizations
PerfGuard
Retrain with enabled jobs

Figure 1: A high level architecture of PerfGuard

a constrained model of query plans [7, 28, 34]. Given the above predictor, we can then identify query plans that require pre-production experimentation. We rely on three key insights. First, recompiling the physical query plan for each job with modifications is incredibly cheap, relative to the actual execution costs. Therefore, we can analyze recompiled versions of physical query plans over large volumes of past workload. Second, while the theoretical space over all possible query plans is prohibitively large, the practical space of plan differences in recompiled physical query plans is far more manageable. And third, predicting the absolute performance of any query plan is a far more difficult task than comparing the relative performance difference between two similar query plans.

The goal is to identify "interesting" jobs (jobs with exaggerated disparities or striking similarities). The remaining jobs will go through experimentation to give us new (real) cost differences and can be fed back into the model before selecting for more "interesting" jobs. We repeat these steps, improving the model, until our budget for pre-production experimentation is exhausted. For the training dataset, we use all previous experiments in the pre-production environments. We can further use recurring query plans with differences over time in order to augment the training dataset.

3 PERFGUARD

We now describe PerfGuard, a system for guarding against performance regressions due to ML-based optimizations. PerfGuard has four components: (i) a data ingestion pipeline to preprocess compile time features, and align them with the physical query plan graphs, (ii) a plan featurizer that calculates a vector representation of the physical query plan, (iii) a module to formulate the difference between the default and modified query plans, and to predict the magnitude of performance regression, and (iv) a module to rank jobs on expected performance regression for flighting.

Figure 1 illustrates the high level architecture of PerfGuard geared for a workload aware query engine that can enable ML-based optimizations for one or more jobs. We denote the runtime performance of the query engine with ML-based optimizations enabled as *New Runtime* (only available in flighting environment) while its compile-time behavior as *New Compilation* (available in production environment). We extract an *Intermediate Representation* (IR) for each workload using the telemetry from the respective optimizer (default and modified) [17]. These IR are tabular in nature and each row in the IR corresponds to a physical operator in the execution plan. Each job is represented using its IR and the execution plan graph, and our first task is to learn a combined vector representation (see plan featurizer in Figure 1). We then calculate the difference between these vector representations. We setup two tasks for learning the plan differences: (i) *regression task* to predict the magnitude of difference along with the direction of performance shift, and (ii) *classification task* to simply predict the direction of performance shift [improve(+) or regress(-)]. Finally, we rank the jobs to produce a subset to deploy the ML-based optimization on and a subset to flight for further validation.

Note that while we focus on performance regression in query processing systems, PerfGuard learns the impact of compile time metrics on performance and could be extended to other domains.

3.1 Data Ingestion

The first step in PerfGuard is to ingest the IR and query plans, align the two representations, and impute any missing values.

IR aligning. The query engine telemetry provides query plans, metadata, and statistics that are collected into an intermediate representation (IR) by the Peregrine framework [17]. IRs consist of 50 different features, including query level attributes (such as a query unique identifier), compile-time attributes (such as estimated cardinality), and run-time attributes (such as job runtime). We also extract DAG representations G_p and $G_{p'}$ of query plans p and p', with and without the ML-based optimizations. We represent the IR as an $m \times n$ matrix where the m corresponds to the number of physical operators in the IR and n corresponds to the number of meta-information on a particular physical operator. Likewise, we represent the DAG of a physical query as an adjacency matrix of size $j \times j$ where j is the number of vertices in the DAG. Not that j and m could be different, leading to a data quality problem.

Missing Data Imputation. There are a number of reasons why an operator could lack one or more feature values, including: (1) logical operators such as spool, sequence, and compute scalar do not have values for run-time statistics, (2) operators such as pod level aggregation could be added or removed dynamically during execution, (3) statistics are not applicable to certain operators.

We can choose to discard the physical operators with missing features from the query plan graph. But this can lead to misconstrued query plan structure. Therefore, we impute the missing features values by computing the cumulative moving average for each column: $\bar{x}_n = \frac{1}{n}\sum_{i=0}^{n} x_i$ that turns out to be quite effective. We also noticed that the JSON query plans represent multiple consumers of a node in the query DAG via duplicated nodes. Therefore, to avoid a bias towards frequent physical operators, we merge duplicate nodes and update the query graph structure accordingly.

Scaling. Scale is important as individual features in the IR differ widely in magnitude. For example, *Estimated Cardinalities* can be thousand of times larger than *Physical Operator Counts*. Therefore, we need to normalize accordingly. We applied scaling on the IR to individual plans separately to provide consistent feature scaling among the plan pairs. This standardization of data also helps us to ensure a faster convergence of our neural network model.

3.2 Model Training

Our primary objective is to measure the similarity between a default query plan p and a modified query plan p' for the same job j in terms of execution cost. One possible approach is to train a model to predict the costs $\langle c, c' \rangle$ of individual query plans $\langle p, p' \rangle$ and use the difference in predicted cost to measure the difference in performance: $\Delta = c - c'$. However, predicting the cost of a query plan for big data systems is not a trivial task [37]. Therefore, we quantify the similarity of two plans directly, which makes the output relative to each input pair. For example, a data shuffle operator (called *Exchange* in SCOPE) is very expensive. If we observe that the modified query plan lacks an *Exchange* operator when compared to the default plan, without knowing the cost of individual plan we can reasonably state that the modified plan is likely more efficient.

To reiterate, we chose to learn the difference Δ_i directly for two reasons. First, we do not compare arbitrary pairs of graphs: p_i and p'_i will always have the same underlying job j_i. Second, the actual cost of running a query is heavily dependent on the physical environment, so even historical data may not be the best predictor [35]. We attempt to mitigate this issue by looking at relative differences. Still, learning only the similarity between $\langle p, p' \rangle$ is inadequate to understand the impact of query plan modification. The modified plan p' may differ greatly from p in terms of query plan structure or run-time metrics, but the difference must be linked to improvement or regression to be useful. We do this by setting Δ as the target variable, where $\Delta > 0$ reflects improvement and $\Delta < 0$ means regression. However, if the scale of the target variable is large, the learning process becomes unstable. As a result, we normalize cost difference for our learning objective: $\Delta_i = \frac{c_i - c'_i}{c_i + c'_i}$

3.3 Job Subset Selection

After calculating the plan difference among the plan pairs, our final task is to rank the jobs. Based on its pair of plans, each job receives a score on a scale from 1 (most improvement) to -1 (most regression). The jobs with the predicted scores close to 1 and -1 are ones that are predicted with high confidence to either improve or regress performance; therefore, they are not considered for flighting. By ranking the jobs on their absolute performance regression score, we can control the number of jobs to flight using a budget k, where lower k value will lead to higher resource savings. Since our objective in PerfGuard is to prevent performance regressions by identifying query jobs that will regress, while keeping the gains of the modifications made to other jobs, we only consider jobs for which our models predicted positive performance shift. However, we still flight them to remove any false positives[1].

4 PLANDIFF MODEL

We now describe our *plandiff* model to quantify differences between query plans. This requires devising a meaningful representation for each of the plans first. We experimented with two different approaches to *plan featurization*. The first is graph convolution based where we learn the plan representations using both IR and the execution graph. The second is manual feature engineering where we create a flat vector representation of a query by aggregating only the IR information of each physical operator and leverage the query structure to provide different weights for aggregation.

4.1 Graph Convolution

Graph Convolution Networks (GCNs) [21] have proved effective for graph representation learning [5, 44]. Intuitively, a GCN considers an underlying graph as a computation graph and learns individual node embeddings by diffusing and aggregating neighboring node attributes across the graph. GCNs have the advantage of being representation invariant; even if the representation differs (e.g. the order nodes in the adjacency matrix), the underlying graph structure remains the same and the GCN learns the same node embedding. The goal here is to learn a function which can provide an efficient node representations in a graph, $\mathcal{G} = (V, E)$, where V is the set of nodes and E is the set of edges. The model takes an $M \times D$ dimensional feature matrix and an adjacency matrix A as input where M is the number of nodes in $\mathcal{G}$ and D is the number of input features for each node. It then produces an $M \times F$ feature matrix where F is the number of output features per node. The propagation rule of a GCN layer is defined as follows:

$$H^{(l+1)} = \sigma[\hat{D}^{-\frac{1}{2}} \hat{A}\hat{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}] \tag{1}$$

Here, $H^{(l)}$ is the output of l^{th} convolution layer, $\hat{A} = A + I$, I is an identity matrix, $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$, $W^{(l)}$ is the weight matrix for l^{th} layer, and σ is a non-linear activation function. The propagation rule $\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}$ denotes the normalization of graph structure. We multiply node properties $H^{(l)}$ with associated layer weight $W^{(l)}$. We adopt GCNs to learn node level coherent feature representations in our work.

GCN Plan Featurizer. Manual feature engineering would require either experimentation or domain knowledge to decide which run-time metrics are most significant. Luckily, GCN-based deep learning models can inherently capture which run-time metrics characterize and influence the intrinsic properties of the plans, and so they are well suited to learn node embeddings. We apply a pooling operation to learn a vector representation of the graph. For example, we can take the mean of node features across the node dimension to formulate the corresponding graph embedding. However, in this process, we lose graph structural information, which is unacceptable in our case. The structural position of a physical operator in the query plan can have a significant impact, ultimately changing the intrinsic properties of the plan. Researchers have proposed different kinds of pooling operations [44] for dimension reduction or node summarization. In our work, we propose an attention based aggregation methodology that we describe below.

4.2 Attention Aggregation

As our primary goal is to learn the similarity between two execution plan graphs, the learned node embedding should be conditional on the given pair. As a result, our aggregation methodology considers the "relatedness" of both the graphs to perform context aware alignment for both the graph structures. To elaborate, let us take a

[1] This ensures that we do not have to learn yet another meta-model.

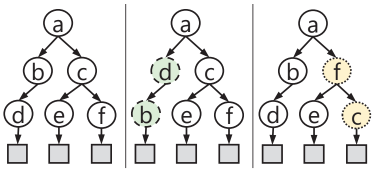


Figure 2: Different execution plans $p, p'\&p''$ of job J generated by different optimizers $Q, Q'\&Q''$ from left to right.

look at Figure 2 where we show two different execution plans (p' and p'') generated by different runtimes Q' and Q'' for job J. We notice that the only difference between plan pair $\langle p, p' \rangle$ is the order of physical operator b and d. Similarly $\langle p, p'' \rangle$ the difference is the order of c and f. Although the changes with respect to default plan p look nominal, it may impact the execution performance significantly. In order to reflect this while calculating the differences between a pair, the node embedding of the modified plan should be conditional on the default plan. Our aggregation methodology focuses on where the plan changes occur.

We denote $S \in \mathbb{R}^{M \times F}$ and $S' \in \mathbb{R}^{M' \times F}$ as the representation matrices of plan p and p', where M and M' denote the number of nodes in the corresponding plans and F is the feature dimension. Our goal is to leverage attention mechanism to obtain p specific representation for each node in p'. We take each node embedding in $s'_i \in S'$ as a query and the node embeddings in $s_j \in S$ as the keys. The idea is to calculate how each node pair $\langle s_i, s'_j \rangle$ influences each other. We compute this attention coefficient as follows:

$$e_{ij} = a(Ws_i, W's'_j) \tag{2}$$

Here, we calculate the importance of modified plan node j to default plan node i. $W, W' \in \mathbb{R}^{\hat{F} \times F}$ are the weight matrices and $\hat{F}$ is the attention feature dimension, and a is the attention function, a dot-product function in our model. We normalize e_{ij} with sigmoid function $\sigma(x)$ to ensure the attention weight is in the range $(0, 1)$.

$$\alpha_{ij} = \sigma(e_{ij}) = \frac{1}{1 + exp(-e_{ij})} \tag{3}$$

We calculate the default plan-centric representation of the modified plan, $Y'_{p'|p} = [y'_1, \ldots, y'_n]$ using weighted sum S': $y'_i = \sum_{j \in M'} \alpha_{ij} s'_j$.

Our next goal is to align the nodes of default plan with respect to the new representation Y' of the modified plan. We calculate an alignment vector for each node in p. Let $A = [a_1, a_2, \ldots, a_M]$ be our alignment vectors for each node embedding $s_i \in S$. Then the aligned node embeddings of p is calculated as follows:

$$a_i = f_a(y'_i, s_i); \quad f_a(x, y) = W_a[x - y; x \odot y]$$

Here $f_a()$ is the alignment function. It is a combination of element wise subtraction and multiplication where $\odot$ denotes an element-wise Hadamard product. The final aligned embedding Y of p is calculated by doing mean pooling over the alignment vector A. After doing attention aggregation, we are left with two context aware plan features for each plan p and p'.

4.3 Model Architecture

Our approach to quantify plan differences is inspired from SimGNN [3], a neural network approach to graph similarity calculations which does not require manual intervention for learning representation. We refer to our modified graph neural network model architecture for plan differences as *PlanDiff*. The fundamental difference between SimGNN and *PlanDiff* is the node embedding aggregation methodology discussed in Section 4.2. First, we produce context aware representations (embeddings) for the corresponding plans by aggregating node embedding matrices using the attention mechanism. Then, by measuring the interaction between the two embeddings using a Neural Tensor Network (NTN) [38], we calculate additional relationships between the plan embeddings, and produce a vector of interaction scores that represent pairs of plans. Finally, to produce a similarity score, the vectors are passed through several fully connected neural network layers.

Our PlanDiff model adopts a similar architecture as proposed by [3]. It involves 3 graph convolution layers, with filter sizes of 128, 64 and 32 respectively, followed by a batch normalization function. Then, we apply our attention aggregator and calculate each individual plan embedding. The plan embedding pairs are then passed through the neural tensor network, which combines the embeddings and generates a "diff" embedding (a vector of size 16). The diff embedding is then passed through 2 fully connected layers. As mentioned previously, query graphs are accurately depicted as DAGs. Despite this fact, we ran our GCN model using bi-directional edges; in using bi-directional edges, we are able to propagate more information about the "neighborhood" of a node in the model.

4.4 Learning Objectives

We employ several loss functions across our various predictive strategies, and enumerate them here while offering brief justifications. We consider two prediction tasks: (i) regression, where we predict the similarity score in a continuous range of [-1,1], and (ii) classification, where we predict a binary value on whether or not there is a performance regression. Overall, rigorously understanding the underlying distribution in the plan difference problem is intractable, and so our choices address this constraint.

(1) For regression, we used **contrastive divergence** (CD) [16], which leverages Kullback-Leibler (KL) divergence to approximate an algorithm's learning gradient. Given a distribution over some vector x, and some weight parameters W, CD is defined as following the gradient of multiple KL divergences: $CD_n = KL(p_0||p_\infty) - KL(p_n||p_\infty)$. CD is an effective way to learn the weights of a network for embedding tasks when we can't evaluate the probability distribution directly, as is the case for our problem.

(2) In the classification framing of our problem, we use traditional **cross entropy loss** function for a classification target.

(3) Furthermore, to assess the uncertainty of the model we use **quantile regression loss** (QRL). Given a prediction y_i^p and ground truth y_i: $QRL = \max\{q(y_i^p - y_i), (q-1)(y_i^p - y_i)\}$ This allows us to parameterize penalizing over-predictions or under-predictions with a specification τ quantile i.e. we can ask our model to avoid over-predictions or underpredictions by nudging it towards a quantile in the unseen distribution.

5 ALTERNATIVE DESIGNS

In the previous section, we discussed featurizing query execution plan and learning difference using graph neural network. In this section we discuss several other approaches to quantify the difference between a pair of plans using the devised plan features.

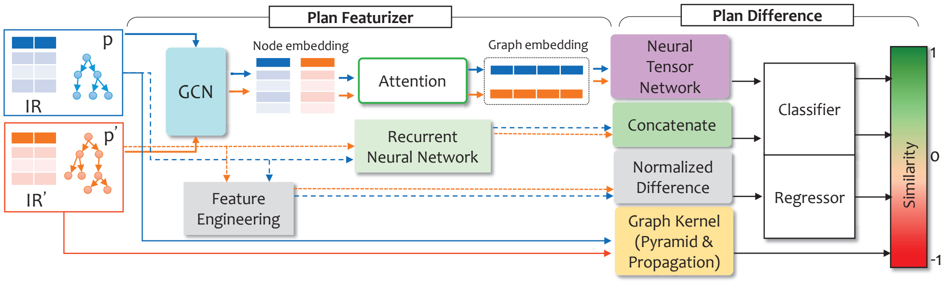


Figure 3: An overview of the algorithmic components of PerfGuard. Solid lines denote the data flow of both query plan graph and associated IR. Dotted lines denote the flow of only IR matrices of the query plans. For *classifier* and graph kernels the output range is $[0, 1]$ and $[-1, 1]$ for *regressor* where < 0 denotes performance degradation and > 0 indicates improvement.

5.1 Graph Kernels

A plethora of graph kernels have been proposed over the years [24] for various tasks like node classification, similarity measurement and missing node predictions attributes, each focusing on distinct structural graph properties. A simple graph kernel compares the *local* structures between graphs: two vertices in different graphs are similar if they have identical labels and similar neighboring nodes. This simplistic attempt to isolate local structures is not suitable in our case. Physical operators in different query plans may have similar neighbors, but the level, or depth, of their location in the graph might differ, and this may lead to significant impact on the query performance. For example, the physical operator *Exchange* incurs a higher cost if it is closer to the root. To address this issue, we consider the global graph structure along with the local substructures when comparing query plans. Our graph kernel must introduce a positional encoding of individual nodes in the kernel method. Therefore, we tested two graph kernels to measure the structural similarity between query plans.

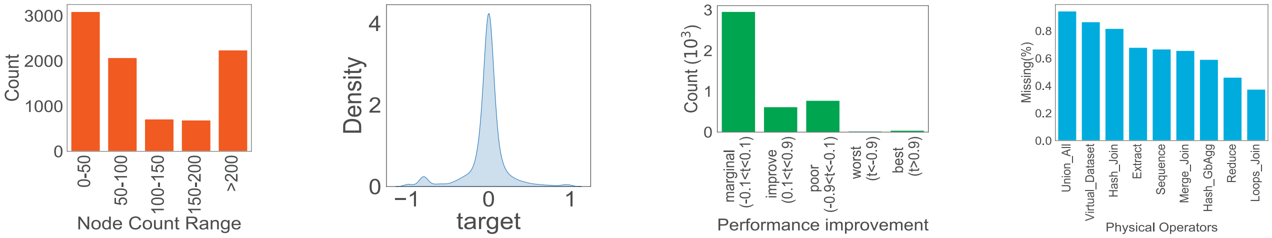
Pyramid Match Kernel [33] compares pairs of graphs $\langle p, p' \rangle$ based on global properties. We focus on the structural similarity between the node labels or operator types, and ignore the IR features first. To achieve this, the kernel first generates an embedding V of vertices by calculating the eigenvectors of the d largest magnitude eigenvalues in the adjacency matrix. The signs of the eigenvectors are arbitrary, so the kernel takes the absolute values of them. The embedding process distills global positional information into individual node embeddings. With the embeddings, the kernel can now partition the feature space into increasingly large regions, and finally project each vertex onto multi-resolution histograms.

Propagation kernel [32] propagates information between nodes based on structure, and the information propagation is often modeled using "Markov Random Walks". We focus on both node labels and IR features in unison to measure the similarity. A propagation based kernel models the graph as a probability distribution $P \in \mathbb{R}^{n \times d}$ where n is the number of nodes and d is the feature dimension of the node attributes. Matrix P_t is the prior distribution of the node attributes, and is initialized with a uniform distribution. We begin our random walk with a random starting node, and the kernel updates the prior P_t with the randomly encountered node attribute values from our walk's path. If T denotes the transition probability matrix during a random walk, then the propagation scheme can be represented with: $P_{t+1} \longleftarrow TP_t$. At each iteration t, we propagate the node information, producing a sequence of graphs. We compare all pairs of nodes from the two query plans $\langle p, p' \rangle$ after each iteration. We aggregate the comparison scores from each iteration after the t_{max} iterations to get the final similarity score.

5.2 Manual feature Engineering

Graph kernels are powerful and provide explicit expressions of the graph, however we find their quality of features in node representation noisy. Therefore, we also consider a manual feature engineering approach using existing compile-time features, similar to [1, 7]. The idea is to calculate different feature channels of a plan for each of these physical operator attributes and then to combine all the feature channels into a vector representation. To encode the structural information, we leverage the query execution plan graph and calculate the maximum depth of the graph and height of each node. We then calculate the weight of a node: $weight = \max(Depth) - height(node) + 1$. The intuition here is that the nodes closer to the root have a higher impact on plan performance. As each row of the IR matrix corresponds to an operator, we multiply each row in IR with the corresponding physical operator weight. We have 32 available compile time features and we only focused on numerical features. It is important to note that feature engineering is done physical operator wise for each plan. This is crucial since the properties/features of physical operators are affecting the query performance. To select the subset of appropriate features we calculated the correlation coefficient of each of the available numerical features. As suggested in [7], we mainly focus on the channels that encompass the important performance related properties with high correlation with the target variable. We consider the feature channels: **1) AvgRowLength:** Length of each tuple **2) Phy opcount:** Count of physical operators **3) InputCardinality:** Total input cardinality from children operators **4) EstCost:** Estimated cost of the sub query and **5) EstCardinality:** Estimated output cardinality from children operator.

The SCOPE engine uses 35 physical operators to forge an execution plan; for each feature channel we initialize a vector of length 35. We concatenate each feature channel and formulate a vector representation of length $35x5 = 175$. We then calculate the joint representation of a pair $\langle p, p' \rangle$ by computing normalized attribute-wise difference per channel. According to Eqn. 3.2 our normalization denominator is the summation of attributes in p & p'.



(a) (b) (c) (d)

Figure 4: (a) Histogram of graph size in node counts (number of phy-ops). (b) Distribution of our target variable. (c) Performance improvement counts of the plan pairs. (d) Missing operator (%) statistics.

5.3 Recurrent Neural Network

Recurrent Neural Networks (RNN) learn representations for sequential data. One major advantage of an RNN is it's power to preserve past information to which the model has been exposed, a crucial requirement in our case. As demonstrated in Figure 2, a small change in the order of how the physical operators appear can have significant impact on query performance. Therefore, we convert the DAG representation of query plans into a sequence of physical operators via pre-order traversal [18]. For example, the left DAG in Figure 2 is transformed into the sequence: $\{a, b, d, c, e, f\}$. Traversing in pre-order fashion ensures that the root node information comes first in the sequence. The RNN then transforms this sequence of node embeddings, and represents the graph by averaging node features across node dimensions. To formulate the plan difference, we simply concatenate the two vector representations of the plans after performing a mean pooling step.

Our RNN model adopts the LSTM (Long Short Term Memory) architecture, due to the strength of its ability to capture both short term and long term appearances of operator sequences. It consists of 3 layers: the LSTM layer with an output size of 32, a fully connected linear layer with an output size of 32 and a final output linear layer. We apply batch normalization after the LSTM layer, and produce a plan embedding after passing the IR sequences through the LSTM. After retrieving the plan embedding for both plans, we concatenate them to formulate the "diff" embedding.

As we are experimenting with multiple different models, we want to see whether we can leverage the strengths of each model by combining their predictions and have better outcome. We applied two ensemble approaches to combine the predictions: *majority voting* and *weighted linear aggregation*. Our majority voting scheme is as follows: if 80% of the total number of models agree on the direction of performance shift, we accept this as a consensus and take the average of the model predictions as the final output - otherwise, the prediction from the *PlanDiff* model is the final output. Our *weighted linear aggregation* weights individual models and calculates a weighted linear sum of their corresponding predictions; the weight calculations come from minimizing the difference between target variable and weighted sum using gradient descent.

In summary, Figure 3 illustrates the model specific components of PerfGuard. The figure outlines different sub steps (featurization and measuring difference) in the modeling process and also exposes how different data structures are being leveraged by different models. Both feature engineering and RNN models operate on IR matrices only. However, the IR rows must appear in a meaningful sequence derived from the execution graph for RNN. On the other hand for feature engineering approach, the order of IR rows does not matter because we are aggregating feature columns. GCN based PlanDiff model and the graph kernels require both physical execution plan graph and the IR matrices.

6 EXPERIMENTS

In this section, we describe the experimental setup for evaluating our PerfGuard framework with real world datasets generated from two successive versions of SCOPE query optimizer. We focus on answering the following questions through experimentation:

- Are traditional graph kernels applicable for our problem?
- How does each individual approach perform at measuring differences among query plans across a variety of evaluation metrics (precision, recall and F1), both for predicting *improvements* and *regressions*?
- How does feature imputation affect overall performance?
- What underlying features and important physical operators does the algorithm focus on when making decisions?
- Are the learned, plan-difference embeddings robust to differentiating among various performance regressions?
- How uncertain are the predictions? (i.e. how far are our predictions from the upper and lower bounds?)
- Reality check: how much time and resources does PerfGuard save when deploying a concrete ML-based optimization, namely learned cardinality [42], on SCOPE?

6.1 Setup

6.1.1 Dataset. We consider a large workload of 4000 SCOPE jobs from the pre-production environment (referred to as *flighting* in Cosmos), where each job ran using two different versions of the query optimizer. For each pair of jobs, we get the physical query plans as JSON objects and the intermediate representation (IR) as tabular files. The difference between the two versions of the query optimizer could be due to a bug fix, new learned features, or any query optimizer enhancement that affects the physical query plans. In our case, differences are caused by enabling learned models for improved cardinality estimations [42]. Figure 4a shows the distribution of the query sizes in terms of the number of physical operators, illustrating the broad range of queries sizes that are considered in our work. Our IR feature matrix includes numerical compile-time attributes, as well as one-hot encoded physical operator names. We normalize the job runtime difference to create the label for each query plan pair. We plot the distribution of our target variable in Figure 4b; it roughly follows a normal distribution, with long tails

Table 1: Metrics for regressions evaluation

Metric	Description
Mean Absolute Error (MAE)	Mean of absolute values of prediction errors
Mean Squared Error (MSE)	Mean of squared prediction errors
R-Squared (R^2)	Percentage of the variance in the dependent variable that the independent variables explain
Spearman coefficient (ρ) and Kendall coefficient (τ)	Measures the correlation between the rankings of two variables
Symmetric Mean Absolute Percentage Error (smape)	$\frac{100\%}{n}\sum_{i=0}^{i=n}\frac{\lvert y_i-\hat{y}_i\rvert}{(\lvert y_i\rvert+\lvert\hat{y}_i\rvert)}$

Table 2: EstCost classification result

	Precision	Recall	F1 score
Improve	0.94	0.49	0.64
Regress	0.06	0.53	0.11

Table 3: Baseline Regression results

Model	MAE	MSE	R^2	ρ	τ	smape(%)
EstCost	0.86	0.86	-12.8	0.0	0.0	92
Pyramid	0.87	0.79	-25.3	-0.1	-0.1	82
PropAttr	0.36	0.15	-4.24	0.0	0.0	70

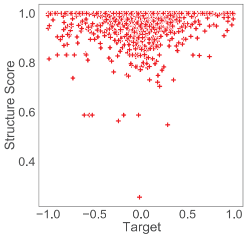


Figure 5: Target variable vs Pyramid kernel score

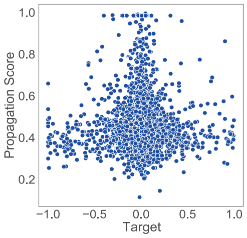


Figure 6: Target variable vs Propagation kernel score

on both sides (with substantial outlier at around the −0.8 value). Furthermore, the distribution is moderately skewed (skewness value of −0.75). Negative target value indicates regression, i.e., the modified query optimizer leads to higher query costs compared to the default query optimizer. Positive target value implies a cost reduction. Data skew is a problem in training regression model and we further confirmed this by looking at the distribution of the residuals which was not a normal distribution. We attempted to address this by re-scaling the target variable using different transformation techniques (log transformation, power transformation etc.), but it did not help. Figure 4c shows the distribution of queries with different performance change (t). Finally, Figure 4d shows the distribution of physical operators with missing runtime statistics.

6.1.2 Evaluation Metrics. We evaluated our model based on the metrics in Table 1. For a regression task, the *Spearman coefficient* (ρ) and *Kendall coefficient* (τ) measure the similarity between the prediction and ground truth. We adopt the scale invariant *Symmetric Mean Absolute Percentage error (smape)* to measure the size of the error in percentage terms. We use the quantile loss function to estimate the lower and upper bounds of our prediction. To summarize, we assess the uncertainty in our predictions by calculating the following three errors: **(1)** $l_e = |lower - actual|$ **(2)** $u_e = |upper - actual|$, and **(3)** $i_e = (l_e + u_e)/2$. Here, l_e, u_e and i_e correspond to lower bound, upper bound and interval errors. We use these measures to understand whether our prediction is closer to lower bound or upper bound.

When framing our problem as a classification task, we evaluate the precision, recall and F1 score of each class. Precision is the proportion of positive accurate predictions, recall is the proportion of actual positives that are correctly identified, and F1 is the harmonic mean of precision and recall. Recall of the *regression* class is particularly important, as our goal is to safeguard our system against performance regressions. In deployment, we cannot afford to miss jobs that will experience performance decline. Aside from these metrics, we calculate the total number of hours saved by not flighting selected jobs to evaluate the job subset selection process.

6.1.3 Model Architecture & Parameter Settings. The tree based models used in experimentation were: *Random Forest (RF), Gradient Boosting (GBR), XgBoost (XGB) & LightGBM (LGBM)*). After the extraction process outlined above, we were left with feature vectors of *length* = 175. The majority of our models are based on boosted trees with the following hyperparameters: a forest of 100 trees with a maximum depth of 3. For the neural network based models, (*RNN & PlanDiff*), we fed the raw IR features directly. We have 61 numerical features in our IR, so the input size for both is 61. We use a *siamese* architecture, which means both the default and modified plan features are passed through the same layers *independently*. We used Azure Standard NC6 VM [2] for running our experiments.

6.2 Using Only Optimizer Estimated Costs

Recall that we use the standard query optimizer at compile time to estimate the cost of a query plan. Using this metric, we can calculate our first baseline prediction: the normalized estimated cost difference (*EstCost*) between the default and the modified query plan. Table 2 shows the classification results with *EstCost*. We simply compare the sign of normalized estimated cost difference and our target variable here. A negative sign implies *regression* and positive sign implies *improvement.* Most of the estimated cost differences indicate performance improvement, which results in relatively high precision for the *improvement* class compared to the *regression* class. This finding suggests that an over-reliance on estimated cost difference would be detrimental for a production environment, as it fails to capture regressions in performance.

6.3 Graph Kernel Results

We consider two graph kernel baselines: *Pyramid kernel* that uses only the query graph structure, and *Propagation (PropAttr) kernel* that uses both the query graph structure and IR feature as node attributes. Figure 5 shows the relationship between our target variable and the structural similarity we calculate using the *Pyramid kernel.* The evidence suggests that a large number of jobs are actually very similar in query graph structure, as most of the similarity values lie between 0.8 to 1 (1 being identical, 0 completely different). In fact, with respect to our target variable, a large number of jobs have very marginal change in performance, and the structural similarity does not appear to play a role in quantifying this performance change. Figure 6 shows the correlation between our target variable and propagation kernel score. We observe that most of the jobs have a propagation score of approximately 0.3 to 0.5. This suggests that even similarly structured plan pairs can have significant differences

Table 4: Performance metrics of regression task for different models with Imputed (I) and Zero Imputed (Z) features.

Algorithm	mae		MSE		R^2		ρ		p@ρ (10^{-3})		τ		p@τ (10^{-3})		smape(%)	
	Z	I	Z	I	Z	I	Z	I	Z	I	Z	I	Z	I	Z	I
Random Forest	0.11	0.11	0.05	0.04	0.26	0.26	0.13	0.16	0.09	0.11	0.09	0.10	0.08	0.6	82	88
XgBoost	0.12	0.11	0.04	0.04	0.25	0.29	0.11	0.18	0.07	0.13	0.07	0.08	0.07	0.3	81	78
Gradient Boosting	0.12	0.11	0.05	0.04	0.22	0.27	0.09	0.15	0.65	0.10	0.06	0.08	0.70	0.2	82	82
LightGBM	0.13	0.13	0.05	0.04	0.23	0.26	0.08	0.18	1.46	0.13	0.08	0.08	1.46	0.4	75	71
RNN	0.14	0.11	0.04	0.03	0.20	0.50	0.13	0.36	0.00	0.00	0.08	0.25	0.00	0.0	73	68
PlanDiff	0.13	0.11	0.05	0.03	0.33	0.49	0.27	0.32	0.00	0.00	0.20	0.22	0.00	0.0	69	68

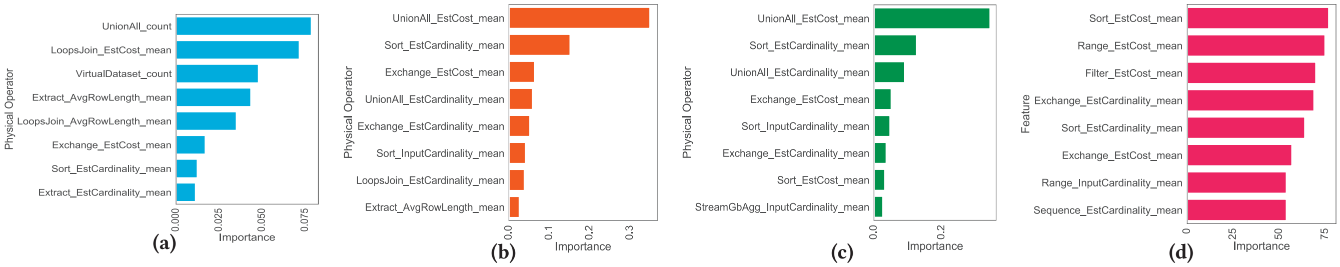


Figure 7: Top 8 important features of (a) XGBoost (b) Random forest (c) Gradient Boosting and (d) LightGBM

in the IR feature attributes, i.e., the compile time settings can differ wildly, regardless of structure. The propagation scores for plans with marginal performance change have large variance, implying that differences in IR feature attributes directly contribute to performance shifts. All of this evidence indicates that identifying and predicting performance shifts in query plans is non-trivial.

We summarized the performances of the baseline models in Table 3. The graph kernels provide positive predictions (0 to 1), so we compared the kernel predictions with the absolute value of our target variable (which can be from -1 to 1) to measure performance. Comparing in absolute values means we cannot confirm the direction (regression or improvement) of performance change when using the graph kernels. Of the three baselines, we observed better performance in terms of *MAE* and *MSE* from the propagation kernel. However, the R^2 scores for all of the algorithms were poor, and the predictions were off by a significant amount, measured by percentage error (smape).

We experimented with graph kernel approaches as part of due diligence, but concluded that they are not the best approach for differentiating query plans based on potential performance regressions. They struggle to factor in adequate meta information about the IR statistics, and produce a seemingly unrelated similarity score to the target variable. In the remaining results from evaluation, we use a standard train-test ratio of 80:20 for experimentation (80% of the data for training the models and 20% for evaluation).

6.4 Regression Task Results

Model accuracy. Table 4 presents performances of individual models for the regression task. Evidently the tree based models, trained with engineered features, exhibit similar performance. Predictions are weakly correlated with the target variable according to R^2, ρ and τ, and the p-values for these coefficients, $p@\rho$ and $p@\tau$, further confirm that weakness. Part of the performance evaluation involves determining the effect of imputing missing values. We do not experience a noticeable performance shift with respect to *MAE* for the tree based models when we impute missing values with a historical moving average as opposed to 0. However, the imputation does change the percentage error *smape* significantly. Surprisingly, the percentage error increases for the Random Forest with imputation. We measure high *MAE* for the LightGBM model, but the lowest *smape* of all the candidates.

To better understand these results, we take a closer look at the predictions from these models. Looking at *MAE* of different models it seems the accuracy of them are very similar. But a difference of 0.01 in *MAE* is very critical in our case as our decision boundary is very narrow. The target values of ≈ 15% of our testing data is in between -0.01 and 0.01. As a consequence a difference of 0.01 can place a plan in opposite region than the ground truth. We observe that most of the predictions for the Random Forest are $\approx -1 \times 10^{-3}$, whereas the median value of our target variable is 6×10^{-4}. We find that 50% of the target values are less than -1×10^{-3}. This suggests that the predictions for the Random Forest experience high *smape* error. In the case of the LightGBM, a lower *smape* hints that many predictions are closer to ground truth than other tree models. The predictions from Gradient Boosting display a very narrow range, producing similar results to the Random Forest, though imputation seemed to improve the R^2 score (unlike the other models). Out of the four tree based models, XGBoost performs best overall.

Both *PlanDiff* and RNN based model perform better than the tree based models. Although these two models have higher *mae* than XgBoost model, but they exhibit higher R^2, ρ, $\&\tau$ scores, and show considerable improvement when missing values are imputed. This suggests that even with higher *mae* the predictions from both PlanDiff and RNN are closer to the ground truth than other models.

Feature importance. In Figure 7 we graph the tree based models' feature importance for the 8 most important features. We calculate feature importance for a single decision tree as the amount that each attribute split point improves performance, weighted by the number of node-wise observations. The feature importance is averaged across all trees, and we find that most top features are associated with expensive physical operators. Other than LightGBM, all the models emphasize the importance of the *UnionAll* operator. Some unimportant operators, like *StreamGbAgg* and *Sequence*, are emphasized by some models. Most of these features relate to the

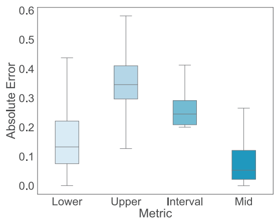


Figure 8: Box plot of related prediction interval errors.

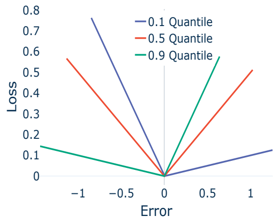


Figure 9: Quantile loss at 10^{th}, 50^{th} and 90^{th} quantiles.

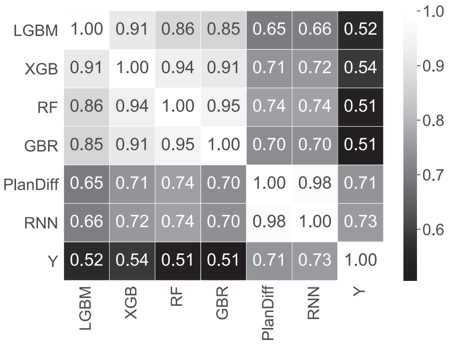


Figure 10: Pairwise correlation between the predictions of different models and a target variable (Y).

difference in *cardinality* (both input and estimated) of the operators, which is strong evidence that the models make decisions based on features that are relevant to the query performance.

Convergence. Figure 11 illustrates the learning curve for these models with 10 fold-cross validation, and utilizes *negative mean absolute error* as the accuracy score (higher the better). In Figure 11a, we observe that the *MAE* of Random Forest jumps to roughly 0.12 after an increased training size (to 100), and both training and validation error appear to flatten post-convergence. A *MAE* of 0.12 is very significant; most of our target values are very close to zero, so an overestimate or underestimate of 0.12 could likely move a prediction to a different spectrum (regression becomes improvement, or vice versa). A high training error for the Random Forest implies that the model suffers from high bias and low variance, and fails to fit the training data well – this is backed up by the high *smape* error as well. XGBoost and Gradient Boosting both have lower training error, and their curves indicate convergence. The higher variance of XGBoost when compared to Gradient Boosting coupled with XGBoost's higher training score means Gradient Boosting is likely more biased. Though Gradient Boosting appears to converge, the predictions exhibit a similar trend to the one we saw with the Random Forest. Interestingly, the training score decreases while the validation score increases initially for the LightGBM, meaning that it overfits at the start of training, but generalizes well.

In Figure 12a and Figure 12b we plot the learning curve for RNN and *PlanDiff*. We trained the *PlanDiff* model for a higher number of epochs, and it incurs a higher training loss and validation loss than RNN. After roughly 4 epochs, both the training and the validation loss flatten. The RNN model takes almost 15 times longer to train, which is why we were only able to train the model for 4 epochs, but we found that both training and validation loss of the RNN decreased significantly within that timeframe.

Pair-wise correlation. Higher R^2 does not always guarantee that a model is doing better, so we also plot a pairwise Pearson correlation for each model prediction along with our target variable (Y) in Figure 10. The tree-based model predictions are highly correlated with each other, and XGBoost displaying a higher correlation on average with the target. Both neural network models are comparatively more correlated with our target variable than the tree based models. This observation is also consistent with comparatively low smape error of both these models. One significant advantage of using an RNN or *PlanDiff* over the tree based models is that it requires no feature engineering. Feature engineering has a high labor cost, and furthermore the type and number of physical operators in an optimizer may change, which would mean recrafting features to reflect those changes individually. The RNN model has some further drawbacks, despite its comparable performance to the graph model; the RNN model takes much longer time to train, and can only process DAGs which leads to information loss.

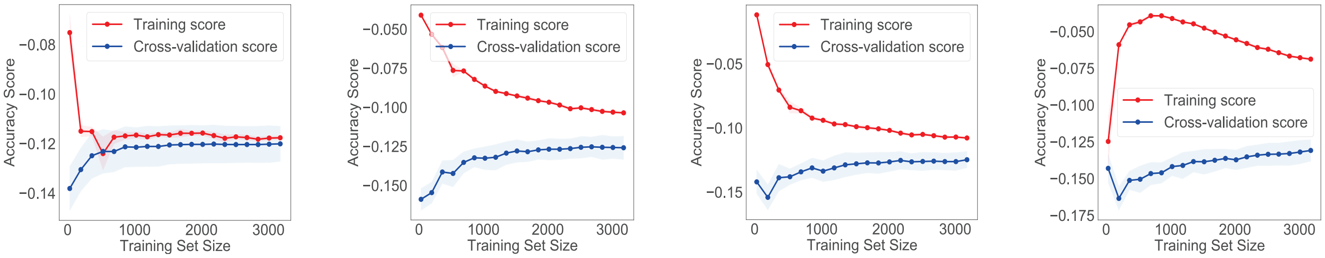
6.5 Analyzing Prediction Intervals

In the previous section we saw how data constraints effect the predictive power of the model. However, we must also consider prediction uncertainty to discern the confidence of the model - in tandem, measures of efficiency and uncertainty provide a holistic model evaluation. We take a close look at the XGBoost model with quantile loss to explain model uncertainty. From our base XGBoost model, we trained two additional XGBoost models with the same parameters with an quantile loss function to compute the upper and lower boundaries for a prediction. Our prediction interval can be thought of as the difference between the upper and lower bound.

First we measured the percentage of data that actually falls within our prediction interval, and discovered that ≈ 85% of the predictions are inside our calculated interval. This percentage alone can be misleading; if the width of the interval is high, most of the predictions will fall inside of it, indicating a high uncertainty in our prediction. By calculating the previously defined Lower (l_e), Upper (u_e) and Interval (i_e) error, we can measure the distance between the actual prediction and the lower and upper bound. Interval error (i_e) is the average of these two distances, and provides an overall picture of a model's total uncertainty.

In Figure 8, the box-plot describes these errors, showing that the median absolute error for the upper bound prediction is higher than median lower bound error. A higher upper bound error indicates that most of the time the prediction is closer to lower bound than the upper bound. This behaviour is desirable: we want our prediction to be near the lower bound to avoid overestimation. However, the median lower bound error is ≈ 0.15, and the median interval error i.e. the width of the prediction interval is ≈ 0.25, which is significant. This implies that predictions near 0 for the upper and lower bounds can indicate a different direction of performance shift. The XGBoost model suffers from high uncertainty.

In Figure 9, we visualize the variance in quantile loss with incurred error for each quantile. The median quantile (50^{th} percentile) is symmetric around zero, while the 10^{th} percentile assigns higher loss to negative errors and lower loss to positive errors. Intuitively,

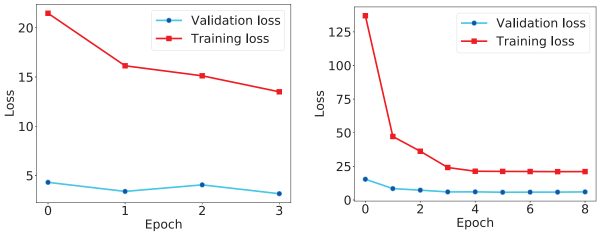


(a) (b) (c) (d)

Figure 11: Learning curves of (a) Random Forest (b) XgBoost (c) Gradient Boosting and (d) LightGBM using cross validation

Table 5: Accuracy, Precision (Pr), Recall (Re), and F1 score of different classifiers with Imputed (I) and Zero imputed (Z) features.

Algorithm	Accuracy[Z]	Improve (%) [Z]			Regress (%) [Z]			Accuracy[I]	Improve (%) [I]			Regress (%) [I]		
		Pr	Re	F1	Pr	Re	F1		Pr	Re	F1	Pr	Re	F1
Random Forest	50	51	48	50	50	52	51	51	55	46	50	53	62	57
XgBoost	52	53	53	53	52	52	52	55	57	47	52	54	63	58
Gradient Boosting	53	54	52	53	52	54	53	53	54	45	49	52	62	57
LightGBM	52	53	50	52	52	55	54	54	56	48	52	54	62	57
RNN	55	53	50	52	52	55	54	57	60	48	52	55	69	61
PlanDiff	54	52	48	50	58	61	60	58	61	49	54	57	69	62



(a) RNN **(b) PlanDiff**

Figure 12: Learning curves of RNN and PlanDiff

the 10^{th} percentile connotes a 10% chance that the true value is below the predicted value and 90^{th} percentile means 90% chance that the true value is higher than predicted. Our results suggest that the algorithm properly assigns a higher error for overestimates.

We further scrutinize each model for the performance intervals defined in Figure 4c. How do the models perform at different intensities of performance shift? Figure 13 highlights the box-plot of absolute error for each model at previously defined different intervals. All of the algorithms perform similarly for query plans with marginal shift, but there is a noticeable performance difference for both RNN and *PlanDiff* for jobs that demonstrate significant improvement. Predictions from most of the models are skewed, with large variability for jobs that experience performance regressions.

6.6 Classification Task Results

As we previously discussed, the task of identifying performance improvements/regressions can be reduced to a binary classification problem. In Table 5, we summarize the accuracy of our models in this classification problem framing, with the expectation that they have higher recall for the *Regression* class. The neural network based models perform much better compared to the models with manually engineered features. In the classification problem framing, data imputation decreases recall and increases precision of the *Improvements* class for most of the models. Just like with the regression task, we inspect the classification performance across different interval "bins." In Figure 14, we present the classification accuracy of the models at each interval. *PlanDiff* and the RNN perform better for jobs with significant improvement – this result is analogous to that which we observed when calculating the same metrics for the regression problem framing. Surprisingly, LightGBM does better on average than most of the other tree models. Overall *PlanDiff* demonstrates balanced performance across different region of the test dataset with higher confidence conforming to Figure 13.

6.7 Results on Spark

Our proposed architecture is engine agnostic. PerfGuard can quantify the similarity of a pair of plans from any big data query system by analyzing a physical query execution plan and/or associated compile-time metrics. To validate PerfGuard's extensibility, we ran experiments using TPC-DS [6] queries on Spark SQL, and performed query optimization focused on learned cardinality models (as we did with SCOPE). Our Spark experimentation produced similar results to our SCOPE experiments, as we detail here.

The graph kernels reported a mean structural similarity score of 0.35 with the *Propagation Kernel* and 0.96 with the *Pyramid Kernel*. We used the same manual feature engineering approach as with the SCOPE experiments (besides *EstCost*, which Spark lacks). The lowest observed *MAE* was the Gradient Boosting regressor (0.03), and all models had a negative R^2. Approximately 43% of the data had target values between -0.30 and 0.30, implying the Spark regression models did not perform as well as the SCOPE models. That our TPC-DS query set is small (only 99 queries), and possibly caused the models to overfit, might explain the worse performance. The high positive R^2 scores for the models on the training dataset and the negative R^2s on the test dataset strongly indicate overfitting.

Hyperparameter tuning did not help to improve our scores on the test dataset. In Spark, the target distribution was heavily skewed and the range of the values were smaller than they were for the SCOPE data (the lowest value was -0.18 and highest value was 0.35). However, some of the models, including the PlanDiff model, performed better in the binary classification setting. In other words,

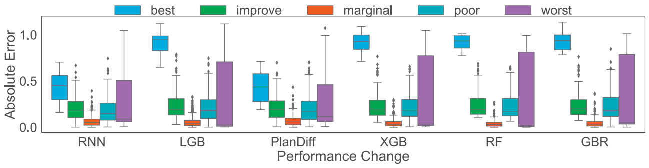


Figure 13: Boxplot of absolute error for each interval defined in Figure 4c.

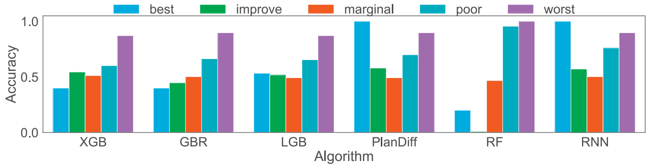


Figure 14: Accuracy of classification of different algorithm in each interval defined in Figure 4c.

some of the models could predict the direction of performance shift, but struggled with predicting the intensity of performance shift.

Gradient Boosting had the highest classification accuracy of 79%, which is significantly higher than what we observed for SCOPE data. However, given the small training query set, any model would likely have lower generalized accuracy. Both the RNN and PlanDiff further demonstrate good results in regression setting with *MAE*s 0.04 and 0.03 respectively. The *MAE* we observed for the RNN and PlanDiff were comparable to Gradient Boosting, but the R^2 was 0.45 and 0.49 respectively, which implies a better fit. Neither model replicated performance in the classification setting – both had lower accuracy than feature engineering based models. Overall, using the best model, PerfGuard is able to correctly predict regressions for 21 out of 30 actually regressing TPC-DS test queries (70% accuracy) when learned cardinalities are enabled in Spark, thus showing the effectiveness of PerfGuard outside of Cosmos.

6.8 Job Subset Selection Results

We now evaluate PerfGuard for our overall goals, i.e., minimizing performance regressions *and* flighting costs. Recall that our primary target is to select a set of jobs to flight based on our budget constraint *B*, where *B* could be the number of jobs or the processing costs. We look at the jobs for which our models predict a positive performance shift and select a $top - k$ subset. If there are N test jobs with ranked cost $C = \{c_1, c_2, \ldots c_N\}$ and top k jobs are flighted then saved flighting cost is defined as: $\frac{\sum_{i=1}^{N-k} c_i}{\sum_{i=1}^{N} c_i}$. The left subfigure of Figure 15 shows the savings in flighting costs over k. Here we consider flighting hours as cost. On the y-axis we chart the ratio of total flighting costs saved and sum of costs of flighting all jobs in our test dataset. Note that as the percentage of jobs to flight increase, *XGboost* and *PlanDiff* save the most hours. For the first 50% of the jobs, the PlanDiff model saves more hours than the other models, and XGboost saves more in the latter 50%. Although both RNN and PlanDiff have moderately high correlations with other approaches in Figure 10, both outperform others. One explanation is that both PlanDiff and RNN correlate more with the target variable, and so more accurately predict the direction of performance shift (if we consider only prediction sign (+/-)). We recommend experimenting both with PlanDiff and XGBoost, our two highest performing models, and note that XGBoost is comparatively inexpensive to train, although PlanDiff performs best overall.

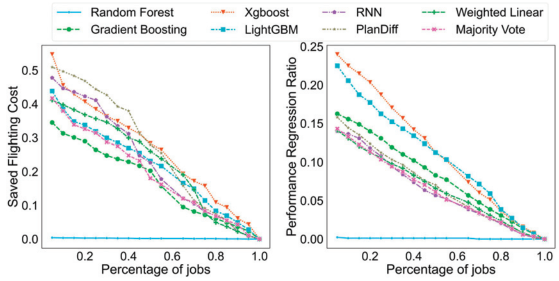


Figure 15: Resource saving (left) and cost (right) of job subset selection using different models.

Only considering total saved flighting costs does not accurately depict the whole picture; there might be jobs for which the ground truth indicates performance regression that we decide not to flight. To address this possibility, we further track the ratio of such cases with respect total number of jobs. If there are p jobs out of k selected jobs which are actually regressing while our algorithm predicted improvement, then we consider them as missed selection and performance regression ratio is defined as $\frac{p}{k}$. In the right subfigure of Figure 15 we plot the progression of missed selections as we increase the number of jobs to flight. Despite saving more flighting costs using XGboost, we risk not flighting a large number of crucial jobs to flight. Both *PlanDiff* and the *RNN*, along with the ensemble models, experience a much lower performance regression ratio. Clearly the *PlanDiff* model saves more costs than the other models, while exhibiting a similar performance regression ratio to the *RNN* and ensemble models. Interestingly the majority vote ensemble model has lower flighting costs along with a similar performance regression ratio to *PlanDiff*; the weighted linear ensemble model performs much better than majority voting. While service provider can vary the number of jobs to be flighted, we recommend to flight 60% of the jobs as a good starting point for Cosmos, resulting in 30% cost savings with only ≈ 5% false positive rate.

7 CONCLUSION

We presented PerfGuard, an end-to-end system that assists in avoiding performance regressions with the new breed of ML-based system optimizations, also referred to as ML-for-systems. We formalized the performance regression problem for big data systems and investigated a number of approaches to correlate changes in the query plans with those in query performance. The key idea is to leverage the large volumes of pre-production workloads and learn from the performance differences of queries that were tested in the past. Our results show that neural network based models perform better compared to models with manually engineered features or models based on graph kernels. The resulting *PlanDiff* model can design efficient experiments that save significant experimentation costs in pre-production and minimize performance regressions in production. We showed the applicability of PerfGuard over both SCOPE and Spark query engines and believe such a framework can also be applied to other domains, such as compiler optimization, in the future.

REFERENCES

[1] Mert Akdere, Ugur Cetintemel, 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.

[2] Azure. 2021. Azure VM NC Series. https://docs.microsoft.com/en-us/azure/virtual-machines/nc-series.

[3] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2020. SimGNN: A Neural Network Approach to Fast Graph Similarity Computation. arXiv:1808.05689 [cs.LG]

[4] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1, 2 (2008), 1265–1276. https://doi.org/10.14778/1454159.1454166

[5] Fenxiao Chen, Yun-Cheng Wang, Bin Wang, and C.-C. Jay Kuo. 2020. Graph representation learning: a survey. *APSIPA Transactions on Signal and Information Processing* 9 (2020), e15. https://doi.org/10.1017/ATSIP.2020.13

[6] Transaction Processing Performance Council. [n.d.]. TPC Benchmark DS Standard Specification Version 1.1. 0 (2012). *URL: http://www. tpc. org/tpcc ()* ([n. d.]).

[7] 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, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1241–1258.

[8] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 969–984. https://doi.org/10.1145/3318464.3389711

[9] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently approximating selectivity functions using low overhead regression models. In *Proceedings of the VLDB Endowment*.

[10] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. In *Proceedings of the VLDB Endowment*.

[11] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 592–603.

[12] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2019. Multi-attribute selectivity estimation using deep learning. arXiv:1903.09999

[13] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *ACM SIGMOD International Conference on Management of Data*.

[14] Rojeh Hayek and Oded Shmueli. 2020. Nn-based transformation of any SQL cardinality estimator for handling distinct, and, OR and NOT. arXiv:2004.07009

[15] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. DeepDB: learn from data, not from queries! arXiv:1909.00607

[16] G E Hinton. 2002. Training products of experts by minimizing contrastive divergence. *Neural Computation 14* (2002).

[17] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload Optimization for Cloud Query Engines. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 416–427. https://doi.org/10.1145/3357223.3362726

[18] M Clara De Paolis Kaluza, Saeed Amizadeh, and Rose Yu. 2018. A neural framework for learning DAG to DAG translation. In *NeurIPS'2018 Workshop*.

[19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[20] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *CoRR* abs/1911.13014 (2019). arXiv:1911.13014 http://arxiv.org/abs/1911.13014

[21] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). http://arxiv.org/abs/1609.02907

[22] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p117-kraska-cidr19.pdf

[23] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 489–504. https://doi.org/10.1145/3183713.3196909

[24] Nils M. Kriege, Fredrik D. Johansson, and Christopher Morris. 2020. A survey on graph kernels. *Appl. Netw. Sci.* 5, 1 (2020), 6. https://doi.org/10.1007/s41109-019-0195-3

[25] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. 2012. Robust estimation of resource consumption for sql queries using statistical techniques. In *Proceedings of the VLDB Endowment*.

[26] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 270–288.

[27] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, M. Alizadeh, and T. Kraska. 2020. Bao: Learning to Steer Query Optimizers. *ArXiv* abs/2004.03814 (2020).

[28] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, M. Alizadeh, T. Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12 (2019), 1705–1718.

[29] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2789–2792. https://doi.org/10.1145/3318464.3384706

[30] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *ACM SIGMOD International Conference on Management of Data*.

[31] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *arXiv preprint arXiv:2101.04964* (2021).

[32] Marion Neumann, Roman Garnett, Christian Bauckhage, and Kristian Kersting. 2014. Propagation Kernels. *CoRR* abs/1410.3314 (2014). http://arxiv.org/abs/1410.3314

[33] Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis. 2017. Matching Node Embeddings for Graph Similarity. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 2429–2435.

[34] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. 1–4.

[35] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.* 3, 1 (2010), 460–471. https://doi.org/10.14778/1920841.1920902

[36] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. 2020. AutoToken: Predicting Peak Parallelism for Big Data Analytics at Microsoft. *Proc. VLDB Endow.* 13, 12 (2020), 3326–3339. http://www.vldb.org/pvldb/vol13/p3326-sen.pdf

[37] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 99–113. https://doi.org/10.1145/3318464.3380584

[38] Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. 2013. Reasoning With Neural Tensor Networks for Knowledge Base Completion. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (Eds.). 926–934. https://proceedings.neurips.cc/paper/2013/hash/b337e84de8752b27eda3a12363109e80-Abstract.html

[39] Ji Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation for Similarity Queries. In *Proceedings of the 2021 International Conference on Management of Data*. 1745–1757. https://doi.org/10.1145/3448016.3452790

[40] Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. 2020. Partitioned Learned Bloom Filter. *CoRR* abs/2006.03176 (2020). arXiv:2006.03176 https://arxiv.org/abs/2006.03176

[41] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2020. Are We Ready For Learned Cardinality Estimation? arXiv:2012.06743

[42] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds.

Proc. VLDB Endow. 12, 3 (2018), 210–222. https://doi.org/10.14778/3291264.3291267

[43] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from Both Data and Queries for Cardinality Estimation. In *Proceedings of the 2021 International Conference on Management of Data*. 2009–2022.

[44] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). http://arxiv.org/abs/1901.00596

[45] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. arXiv:2006.08109

[46] 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. arXiv:1905.04278

[47] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *VLDB J.* 21, 5 (2012), 611–636. https://doi.org/10.1007/s00778-012-0280-z

[48] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2020. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. arXiv:2011.09022