# OS Pre-trained Transformer: Predicting Query Latencies across Changing System Contexts

Parimarjan Negi[1], Ziniu Wu[1], Arash Nasr-Esfahany[1], Harsha Sharma[1], Mohammad Alizadeh[1], Tim Kraska[1], Sam Madden[1]
[1]MIT CSAIL
{pnegi,ziniuw,arashne}@mit.edu
{harsha,kraska,alizadeh}@csail.mit.edu

## ABSTRACT

Current approaches to instant specialized ML models can learn strong query latency predictors for a database in one system context (e.g., AWS instance type with a particular load). However, such models fail catastrophically when tested on a new system context. This generalization gap requires training data of each database workload for every new system context. We propose learning independent models for how the database and the system context affect performance. We introduce OS Pre-trained Transformer (Osprey) that takes recent time-series of OS metrics, such as CPU and memory usage, and produces an embedding vector that represents useful performance properties of the dynamic system state. Osprey is pre-trained on a large set of workloads to learn a universal model of the OS log measurements. This is used by us in a novel factorized model architecture to predict DBMS query latencies. This allows us to train a query plan model specialized to a particular database on just *one system*, and get generalization to several different class of systems, and system loads. On new system contexts, our model is up to 3× better at the median and mean latency errors than a predictor trained on the same system as us. In fact, our model is as accurate on new systems or loads as if we had trained the instant specialized model just for that system. This can be done because (1) we cleanly separate out the effects of the system context v/s the query specific effects, and (2) we pre-train the Osprey model on many other databases across various system contexts in order to learn how query latencies vary as system resources vary. The datasets and code are available at: https://github.com/parimarjan/LatencyPredictor.

## 1 INTRODUCTION

Modern DBMSes are moving from local servers to cloud native architectures [41, 42, 46, 49, 52, 57]. This has several benefits, but also leads to complexity which makes it harder to adapt traditional analytical cost models for query performance prediction (QPP) [42, 49, 52]. There are hundreds of available hardware configurations which vary resources such as I/O backends, CPUs, memory, network speeds. Moreover, due to virtualized (shared) hardware or multi tenancy, performance can vary over time on the same hardware.

As an example, we execute an IMDb workload [43] on ten different AWS instance types (Figure 1). The workload's average runtime goes from 7 seconds to over 200. Several instance types also show variance in execution of the same query — this shows typical variability in the cloud. In response to this complexity, machine learning methods have been proposed that learn QPP models in a data driven way. This is attractive because the transition to cloud architectures has allowed DBMS providers to collect logs and telemetry at an unprecedented scale, with analysis on millions of daily jobs reported by Snowflake [60], AWS [49], and Microsoft [42, 52].

| Instance Type | Mean | Per Query Std |
|---|---|---|
| **a1_large_gp3_4g** | 78.2 | 13.9 |
| **c5a_large_mag_4g** | 204.8 | 11.7 |
| **c7g_large_mag_4g** | 195.6 | 4.8 |
| **m6a_large_mag_8g** | 70.5 | 15.9 |
| **r6a_large_mag_16g** | 7.3 | 1.7 |
| **r7g_large_gp2_16g** | 9.13 | 0.59 |
| **t3_large_gp2_8g** | 34.9 | 43.1 |
| **t3_xlarge_gp2_16g** | 11.8 | 2.1 |
| **t3a_medium_gp3_4g** | 69.4 | 17.3 |
| **t4g_large_mag_8g** | 72.3 | 13.5 |

**Figure 1: Average query latencies, and std, for 6 repetitions of** 320 **queries from an IMDb workload on AWS. Instance Type is 'aws instance name'_'I/O-backend'_'RAM'.**

Instance optimized models are a widely adopted approach for such models in both academia [14, 32, 38, 39] and industry [49, 61]. These are optimized for a specific scenario, e.g., latency prediction on a database in a fixed execution environment. For the example in Figure 1, following the approach in [38, 49] would require ten separate models for the IMDb workload on each instance. Instead, we propose a factorized model architecture that requires a single instance specialized model that can learn about the current workload in one system, and have a pre-trained universal model of how system state affects query performance, which adapts the instance optimized model to changes in the system context.

Our system model is OS Pre-trained Transformer (Osprey). Osprey takes as input time-series measurements of recent history of system metrics, and produces an embedding vector to capture various system factors influencing performance in a common embedding space (Figure 2). Osprey is pre-trained on system logs from a large set of past workloads to learn the factors most relevant to predicting query performance. Finally, we use a latency prediction module which learns how query performance varies with changing system contexts represented in this embedding space.

The key benefit of our approach is that it separates out two independent factors affecting query performance: (1) estimating workload specific properties (query complexity), (2) system specific properties (hardware and system load). Instance specialized models [38, 39, 49, 67] learn both these aspects together, making

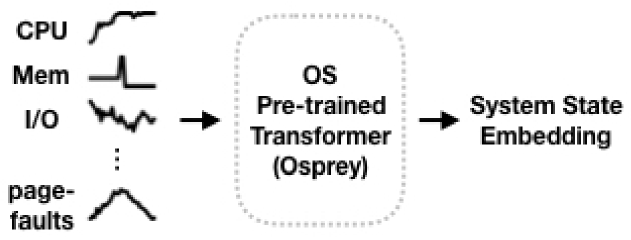


**Figure 2: OS Pre-trained Transformer takes recent time-series data of system log measurements and produces an embedding vector which is useful for downstream tasks that depend on the system state.**

them very brittle to any change in execution conditions. We find that instance optimized models are needed in understanding the first aspect, as workload patterns can vary arbitrarily from one workload to the next. This is not true when considering the system context's impact on performance. Unlike query plan estimates, system state (e.g., memory utilization or I/O speeds) can be precisely measured and meaningfully compared across different hardware and workloads.

Osprey has the benefit that it treats both hardware changes, and other changes that affect system load such as multi-tenancy, or virtualized hardware, in the same unified manner. More crucially, it allows us to leverage data from any workload to learn patterns showing how these metrics evolve over time. It is inspired by how large scale pre-trained transformer models have revolutionized all tasks in Natural Language Processing [13, 47] and Vision [48]. The core idea behind these techniques are not restricted to text or visual data, and similar efforts are being made in several fields. We believe the following key properties suggest that OS level logs would be a great candidate for large pre-trained models that can be useful for many downstream tasks.

1. **Presence of large scale datasets.** Most of the benefits of pre-trained models became apparent when trained on internet scale data. For OS's, system logs are constantly being generated by all applications. For query processing, the existence of large scale datasets is shown in surveys by cloud providers [49, 60], however, we are not aware of any work that trains a universally useful model at a large scale on OS logs.
2. **Right level of abstraction.** OS state is central to any systems application, therefore, we believe representing its properties in a unified vector space can be useful for many downstream tasks from scheduling systems, to workload classification, to performance prediction.
3. **Well defined downstream tasks.** While its intuitive that OS state is applicable to many downstream tasks, there are not many standard ML for systems benchmarks. Therefore, one of our goals was to explore one such downstream task in depth in order to create a well defined benchmark and pave the way for other such efforts.

We show Osprey pre-trained on a large set of workloads learn embeddings that are very useful for latency prediction on completely new workloads. For a new workload, we train an instance specialized query model on just one system; then using our pre-trained models, we can get accurate predictions on any other systems known to Osprey without retraining the query model. These predictions are as good as if we had trained an instant specialized model on that particular workload, system pair, and up to 3× more accurate than instance specialized models which were trained in different systems. Further, we find additional benefits of Osprey: (1) it's OS state embeddings can cluster different workloads by their performance properties, which can be used to find similar workloads or analyze workload distributions (2) Osprey's internal mechanisms can be used to highlight which system metrics from almost a hundred are most relevant for performance prediction in the current OS state.

Our main contributions are:

1. **OS Pre-trained Transformer.** We propose a pre-training approach that lets us learn the dynamics of a system environment from several workloads. Our results are a proof of concept at a small scale — but the same approach could be extended to significantly larger set of commercial workloads.
2. **Factorized model.** We combine the query plans, and the system logs, into a single model architecture that uses a Graph Convolution Network (GCN) [29] for query plans, and a pre-trained Transformer [58] network, Osprey, for processing the system logs.
3. **Dataset.** We construct a dataset by executing the same queries in various system contexts to highlight the variance, and develop new techniques that take the query's environment into account.
4. **Parametrizing the system state.** We believe that most system state effects are explicitly, or implicitly, captured by OS system metrics. Therefore, we propose that ML systems performance prediction work should consider such metrics as an important feature.

## 2 RELATED WORK

Our work seeks to learn how the system state affects query performance in order to use a learned predictor model across various hardware and system states. There has been a lot of work on related topics, so we will use this section in order to give a broad overview of the field, and highlight differences with our approach.

**Query Performance Prediction (QPP).** Predicting execution times of queries has a long history in research for DBMSes. Classical approaches carefully model query plans, and use hand tuned cost models for different hardware [6, 7, 15, 20, 25, 30, 36, 51, 55, 63], which were optimized for the on-prem DBMS server settings. Several studies from cloud providers have shown that such cost models don't easily translate to the cloud DBMSes: such as DBMS costs not correlating to observed runtimes in Microsoft's SCOPE [42, 52], or learned models clearly outperforming DBMS costs in AWS Redshift [49].

**ML based QPP.** In order to address such challenges, a lot of recent research has considered learning query performance directly from past workloads using ML approaches [2, 4, 8, 9, 22, 22, 23, 39, 40, 42, 49, 54, 59, 64, 65, 67, 69]. These include several kind of models: per operator models [2, 52], query plan models [9, 22, 23, 38, 39, 54, 67], or models that flatten query plans to key features [42, 49, 66]. All these papers focus on representing a query plan to make latency predictions. None of these consider the effect of the system state explicitly. Instead, they assume the system state remains consistent between the training and test phases — thus, a new model would need to be trained for every <database,system> pair. Such a system was implemented in Redshift [49], but they also describe its challenges: (1) nothing globally useful is learned despite having millions of queries (2) a learned model is only able to start predicting after seeing hundreds of queries on a new instance. Moreover, such a model does not help in answering 'what if' questions, for e.g., what if we increased the memory capacity of a server?

**QPP across different hardware and multi-tenancy.** Classical DBMS approaches systematically analyze the effect of hardware on operator costs [50], and develop analytical cost models to account for such changes. There has also been a lot of research on how changing specific hardware components affect DBMS performance, e.g., Amdahl's law for multi-processors [3], effect of different I/O compnents [24], memory allocators [17], and so on. However, these are not easily combined to develop analytical models for how performance changes across different hardware or system states. In one recent paper, Ji et al. [26] create a causal graph between multiple system resources, like Memory, I/O etc., and the DBMS query performance, and use this to identify root causes for anomalously bad query performance. However, there isn't much work on data driven methods that learn the interaction of different resources in order to accurately predict query performance. Moreover, this problem becomes even more relevant in cloud architectures that use multi-tenant architectures [31, 60], since different tenant workloads can arbitrarily change the system load. Our approach addresses this gap in research, and we show that we can deal with both these scenarios in a unified manner via measuring the key metrics that relate to system state.

Delimitrou et al. [11, 12] study datacenter job completion times, using matrix completion methods to estimate latencies on new hardware, which is conceptually similar to our work in disentangling query and system state effects on latency. However, there are several practical differences, e.g., they rely heavily on sampling of homogenous subparts of the job, similar jobs, and do not explicitly model the OS system state.

**Concurrent QPP.** Another related research area is predicting performance of concurrent queries [16, 62, 68], i.e., a set of queries that would execute simultaneously on the same database. This involves accounting for both the strain on system resources and potential efficiencies from shared scans in memory. Our work, however, primarily aims to differentiate the effects of system state from database-specific impacts. Concurrent QPP adds the complexity of query interactions on the same database, and addressing this is beyond our current scope.

**Pretraining universal models.** Since there is so much workload data available in the cloud [42, 49, 52, 60], it is natural to ask if we can train universal models that learn using the whole dataset. Lu et al. learn pretrained summarization models across many database workloads that can be quickly adapted to cardinality estimators per database [35]. This is similar conceptually to how we learn pre-trained OS system dynamics model for latency prediction, which is then adapted to instance specialized per database models. We are not aware of other work in this direction for latency prediction. There have been some approaches to learn universal query plan models for latency prediction [23, 27, 64], but this remains an active area of research, and as argued in a study of a Redshift deployment [49], in practice, there are several challenges to have a query plan model that generalizes to any database. Therefore a lot of research focuses on instant specialized systems that are trained for just one workload [14, 38, 49].

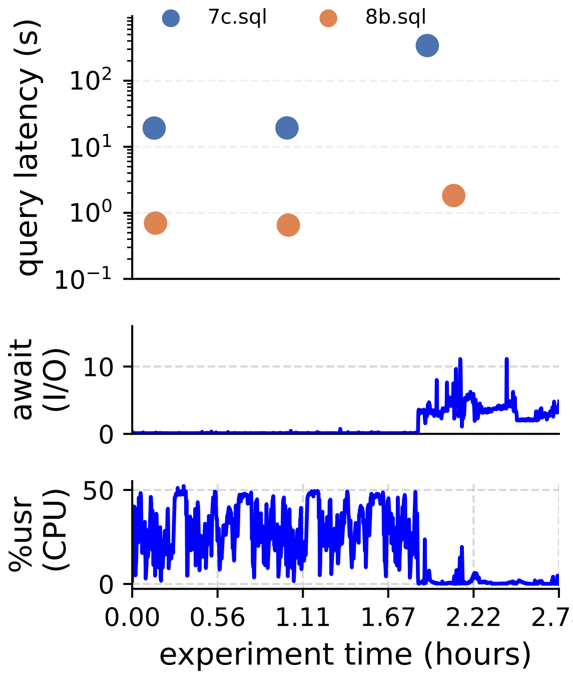


**Figure 3: Running IMDb workload, one query at a time, on r7xlarge (gp2 I/O backend). The variance in runtime is only due to varying I/O speeds of the gp2 backend.**

In a recent vision paper [49], Saxena et al. lay out a description of foundation models for operating systems, and discuss several different downstream applications such as caching or query performance. Conceptually, our work is in the same direction, and we go deep into one such application (QPP) of pre-training models on large amounts of operating systems data.

## 3 QUERY LATENCIES WITH CHANGING SYSTEM CONTEXTS

Consider the query latency scatterplot in Figure 3 which execute queries from IMDb workloads [34, 45] on a r7xlarge instance with gp2 I/O backend. Only one query executes at a time, and we run three repetitions of each query. We can see that the same query, 7c.sql, has up to 10× difference in execution times on the same

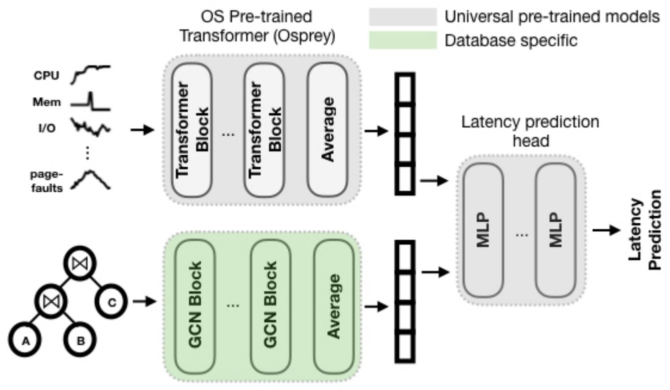


Figure 4: Factorized model architecture.

system after about two hours. This is due to the system state. The AWS gp2 I/O backend varies I/O speeds over time, sometimes giving faster speeds called I/O bursts. In such cases, ML models which only consider query plans [23, 38, 39, 67] can not do better than predict the average latency of the three executions. This implies they can have large errors when either the training or test data is from dynamic scenarios.

How would a performance engineer predict query performance in such settings? They may benchmark the query, or look at the query plan and statistics to get an idea of how much work needs to be done, and what are the critical operations. Next, they may consider the new system — I/O speeds, CPUs, available memory. Maybe run 'top' or other utilities to look at the load on the system. Finally, combining these sources of information, the engineer would have a sense of the expected execution time of the query. We want to do this automatically. We design our model to make a prediction integrating these different sources of information. They can handle such dynamic cases well by also considering the system state, in the form of time-series data for recent system logs.

Let us consider the examples in Figure 3 again to see how this may work concretely. The varying I/O speed of the system is reflected in the logs for 'await' (the average time an I/O operation takes from the moment it's issued until it's completed). The effect of the lower I/O speed is also seen indirectly on the lower cpu utilization shown in the plot for '%usr'. At the same time, not every query would be affected as severely by the slower disk I/O — 8b.sql still runs quite fast, because it was not doing as much I/O in any case, which shows the importance of modeling the query plan as well as the system logs.

This was a simple example. In reality, we provide time series data on 92 such system log measurements[1] to our model, which should allow it to learn the effect of more complicated patterns. As an example: few page faults, high memory usage, and high cpu usage might mean that the accessed database is

## 4 OUR MODEL

**Problem statement.** Our system takes in a query plan, and OS logs representing its recent state, and predicts the query latency. The latency can be affected by various factors, like query complexity, or the system load, or the hardware.

**We assume a steady system state.** An important assumption is that the system state does not change drastically just before and during query execution, i.e., we aim to make the best potential prediction using information at query compile time. Our prediction would be completely wrong if the system becomes overloaded after the query starts. This is reasonable for queries taking seconds to minutes, as system load is typically stable in such intervals.

### 4.1 Factorized Model Architecture.

Several past works have focused on the architecture and featurization of query plan models for latency prediction [2, 23, 39, 49, 59, 64]. Our contribution is in proposing a factorized model architecture which expands on the query plan model to be adaptable to different hardware and system states. It includes three distinct models as shown in Figure 4.

(1) **Database specialized query plan model.** This can be any neural network based query plan model (Graph Networks [8, 22, 23, 23, 53], Tree Convolutional Networks [38, 39], or others [67]) which takes as input a query plan and is trained from scratch on every new database workload. Past work typically used such models to predict latencies directly, but we want the output of this model to be usable for predicting latency across many system contexts. Therefore, our model produces an embedding vector which should contain all query specific properties required to predict latencies.
(2) **OS Pre-trained Transformer (Osprey).** A universal (i.e., applicable to any workload) transformer model which takes as input recent OS logs, and produces an embedding vector that captures the system state. It is pre-trained on a large set of old workloads that cover the space of system states, and after pre-training, its weights are fixed. On a new database, we will only train the DB specialized model, and Osprey will act as a fixed function.
(3) **Latency prediction head.** The information from the query and environment embedding vectors are used as input to a universal Multi Layer Perceptron (MLP) module to produce the latency prediction. This approach parallels the training of task-specific heads atop a commonly learned representation in natural language processing (NLP), exemplified by models like BERT [13] and GPT [47], which are fine-tuned for specific tasks, such as sentiment classification [19].

**Database features use estimates, therefore, we use database specific models.** Various studies have proposed instance specialized models for specific databases [38, 39, 49]. A key reason for this specialization is that query plan features rely on DBMS estimates, such as cardinalities and costs. However, these estimates are inaccurate, sometimes by orders of magnitude. Database-specific

[1]Examples can be explored at: https://parimarjan.github.io/latency_collector/Query-System-Logs-a1_large_gp3_4g-imdb.html

models can learn to fix the cost model errors introduced by these estimates by identifying patterns, such as correlations between specific columns in the workload. Training a model across thousands of databases, however, will prevent it from leveraging such unique workload patterns.

**Universal pre-training with environment features is based on precise measurements.** System logs provide precise measurements of relevant metrics, enabling the comparability of their time-series evolution across various workloads. This generality of metric evolution makes a compelling case for using a universally pre-trained model. The rationale is that the influence of environmental factors, such as the number of CPUs or the amount of memory, can be understood in a generalized context. For instance, workloads that demand extensive processing could benefit from additional CPU resources. Similarly, memory or I/O intensive workloads might show improved performance with enhanced respective resources. This principle of environmental impact is not workload-specific but rather a universal aspect that can be learned and applied broadly.

**Generalization properties.** There are two dimensions along which our predictions would need to be robust: (1) query properties (2) system states. A lot of past work studies robustness of deep learning based query-driven models to new kinds of queries [28, 38, 43, 44, 64]. Our model should work well in such cases where the new queries are from a similar workload as the training set. The pre-trained model adds robustness to changing system contexts, which is our key new contribution.

**Independence of query and environment models.** One important aspect of our model is that the query and environment model learn independent representations. In particular, for a given query, the query model will learn the same representation across any environment; similarly, the environment model will learn a representation that is useful for predicting latencies of all queries in similar system states.

## 4.2 OS Pre-trained Transformer

Here, we describe our approach to tokenizing the system state, and the architecture and training of the OS Pre-trained Transformer model used for generating the environment embedding vector.

**OS logs as system state.** We use the linux sysstat [1] package to collect periodic system measurements throughout the experiments. The package provides several log files, and the ones used by us are listed in Table 1. We do not use an additional 20 log files that are specific to various networking sub components, that are not relevant to our specific workloads. We choose these log files because these measurements capture the dynamically evolving state of the system in a way that is comparable across different linux based systems. Moreover, there are several robust and efficient packages tailored for capturing such measurements, which streamlines the data collection process and ensures consistency. In general, we can have an arbitrary number of log files (e.g., if we wanted a log file for each I/O device or CPU), since the transformer architecture supports arbitrary length sequences.

**Tokenization.** The time-series data of each log sub-field is turned into one token for our model. Each token is the previous $N$ measurements, and an identification vector indicating the metric. In our experiments, we choose $N = 30$ with measurements collected at 20 second intervals.

| Log File Name (Flag) | Included Fields |
|---|---|
| Paging Stats (-B) | pgpgin/s, pgpgout/s, fault/s... |
| I/O Rate (-b) | tps, rtps, wtps... |
| Block Device (-d) | tps, rkB/s, wkB/s... |
| Filesystem Usage (-F) | MBfsfree, MBfsused, %fsused... |
| File Table Status (-v) | dentunusd, file-nr, inode-nr... |
| Hugepages (-H) | kbhugfree, kbhugused, %hugused... |
| Interrupts (-I) | intr/s |
| Power Mgmt (-m) | MHz, rpm, DEVICE... |
| Memory Usage (-r) | kbmemfree, kbavail, %memused... |
| Swap Space (-S) | kbswpfree, %swpused... |
| Task/Switching (-w) | proc/s, cswch/s... |
| CPU Utilization (-u) | %user, %nice, %system... |
| System Load and Pressure-Stall (-q) | %scpu-10, %sio-10, runq-sz... |

**Table 1: Summary of SAR log files and example subfields.**

**Transformer model.** The transformer architecture [58] has emerged as the most popular general purpose ML architecture for sequence to sequence modeling. We use a decoder only transformer architecture as described and implemented in [5].

Let $L_i$ be the $ith$ token characterizing the recent time-series evolution of a metric. The function signature of one block of the transformer architecture will look like:

$$T(L_0, L_1, ..., L_n) = Emb_0, Emb_1, ..., Emb_n \tag{1}$$

i.e., takes in a sequence of log measurements and transforms them into a new sequence with the same shape, which goes into the next transformer layer. After the transformer blocks, we average these sequences to get an environment embedding, as shown in Figure 4. Each transformer block consists of multi-head attention, feed-forward MLPs, residual connections, and layernorm — see Bloem [5], Vaswani et al. [58] for detailed descriptions of the architecture. Here, we will focus on the attention mechanism, which is a key component of the transformer architecture in order to develop intuition about what the model may be learning. Each block uses several self attention modules (multi-head attention), but for simplicity, we'll describe the computation for a single self attention module:

$$Emb_i = \sum_j a_{ij} \cdot f(L_j) \tag{2}$$

i.e., $Emb_i$ is a weighted sum over a (learned) function of each the input logs ($f(L_j)$). The weights, $a_{ij}$, or the 'attention scores' sum to 1 over all $j$. The key aspect of self attention is that $a_{ij}$ is a function of the input logs $L_i$ and $L_j$.

Thus, each $L_i$ is transformed depending on its relationship to the system context; in this case, the rest of the system log metrics. This is a key aspect of Osprey: it considers the full input sequence (all the system logs), and reweighs the importance of different sub-parts.

For instance, given $L_i$, if all $a_{ij}$ are equal, then $Emb_i$ would just be an average of all input logs; On the other hand, if the model considers some $L_j$ (e.g., CPU utilization) to be most critical, then the output $Emb_i's$ can depend more on that if $a_{ij}$ is larger. Or, if the model considers the interplay between memory usage and page-faults to be the most relevant, it would extract embeddings that depend most on these two log measurements. Therefore, the attention matrix containing these $a_{ij}$ scores is often used for interpretability as it shows the input relationships most relevant from the model's perspective.

**Pre-training task.** In order to train the Osprey and the final latency prediction head, we would use query latency and system logs from a large collection of databases and system contexts. All three modules — query model, Osprey, and prediction heads — will be trained together to minimize the prediction error over the full dataset. Since the query prediction model was specialized to this particular set of databases, it will not be useful for a new database; therefore, we will discard it. The Osprey and prediction head will be fixed after pre-training; their input does not depend on a specific database, and can be used as a fixed function.

**Training on new databases.** For a new database, we will initialize a new database specialized model with appropriate featurization. This will be trained from scratch using the same architecture as Figure 4 — with the Osprey and prediction head acting as fixed functions. The new training workload may only be from one system context, but the new DB model will need to learn to produce query embeddings that are compatible with the universal models pre-trained earlier, which increases robustness of latency predictions to the many other system contexts seen during pre-training.

### 4.3 Discussion

Here, we would like to discuss various benefits and challenges of our approach, and its potential for real world scenarios.

**Model size and scalability.** In all our experiments we use 4 transformer layers, and 16 heads per layer, an embedding size of 32. Also, we used system history of 30 time steps at 20 second intervals. We chose these settings as we found them to perform well on our workloads. This creates a model with $3M$ parameters, which is a very small model compared to modern pre-trained transformer architectures [47]. But these are proof of concept experiments, which would only work in the system contexts in our dataset — i.e., 10 AWS hardware types, with particular kind of loads. If we scale up such architectures to larger scale datasets, we will likely need to significantly scale up the model sizes as well. As trends from the broader ML communities suggest, the transformer architecture is particularly good at scaling to large scale datasets, which suggest that such an approach should scale well to massive real world workloads as well.

There are several benefits of processing OS logs using a transformer based architecture to capture the system state:

(1) **Lightweight.** There are several well tested measurement libraries for system logs which are lightweight; in particular, using special hardware primitives such as perf-counters [10] adds even more efficiency to these measurements.
(2) **Relevant scale**. In our experiments, and also in published papers from cloud DBMS provides [42, 49], query latencies last from a few seconds, to over several hundred seconds. These OS measurements typically do not change drastically in the scale of seconds, which is relevant to predicting query latencies.
(3) **Amortized inference.** In general, the transformer architecture is about $5x$ slower than the query part of our architecture (on CPUs) — adding tens of milliseconds to each inference. However, the transformer model is independent of the query — thus the environment embedding does not change over our sampling frequency. Thus, the inference costs of the model can be amortized over several queries.

However, there are also several drawbacks that need to be overcome to make such an approach practical.

(1) **Logging overhead.** Many system metrics are not logged within query plans, requiring separate logging solutions. Also, the increased volume of logging data may pose data management challenges at scale.
(2) **Variability.** Different systems and cloud providers may care about different metrics.
(3) **Distributed DBMS.** In distributed systems, its more complicated to collect logging data from all system sources that can affect performance.

## 5 DATASET

In this section, we describe the dataset we collected, which includes over $150K$ PostgreSQL query execution plans and corresponding linux system logs for over 2500 unique queries from 14 database workloads executed on 10 AWS instance types, and under various system loads. The dataset is designed to study how the changing system environment affects query performance. This problem has been studied since 1980s [50]. However, earlier research was focused on designing DBMS cost models, and targeting smaller scale setups, more typical of on-prem settings. In contrast, cloud providers collect and analyze logs from massive workloads [42, 49, 52, 60]. While it is impractical to collect a dataset comparable to commercial scale, we collect a non-trivial-sized dataset that shows execution time variance in modern cloud scenarios.

Table 2 summarizes some of its key properties. The workloads are all based on publicly available benchmarks used in past work on ML-based latency prediction [23, 38]. The database sizes range from $< 1GB$ to $100GB$. One key property of our dataset is that each query is executed on ten different AWS instances, with varying system loads — which allows us to analyze the effect of the system environment on the query. Each query is repeated up to three times in each system context — which gives up to 100 repetitions of each unique query.

We show the average standard deviation of query latencies (in seconds) in a workload across the 10 AWS instances with one query running at a time, denoted as "Latency Std (Hardware)" in Table 2. We observe that some of the workloads, e.g., IMDb, Stack, and TPCH, are greatly affected AWS instance type change while other workloads, such as the smaller $1GB$ RelationFit workloads are not affected as much. This is because the smaller DBs can fit in the RAM

| Workload | DB Size | #Unique Queries | Latency Std (Hardware) |
|---|---|---|---|
| ***CEB [38, 43]*** | | | |
| Stack-CEB | 100GB | 240 | 8.9 |
| IMDb-CEB | 7GB | 320 | 12.6 |
| ***RelationFit [23]*** | | | |
| ssb | 1GB | 150 | 1.6 |
| ccs | 1GB | 150 | 0.4 |
| accidents | 1GB | 150 | 0.7 |
| *...and 7 other database workloads with very similar statistics* | | | |
| ***Others*** | | | |
| TPC-H (S=10) | 10GB | 22 | 24.6 |
| Stats (Alibaba) [21] | 10MB | 140 | 0.1 |

**Table 2: Workload details. Queries are executed under different conditions on 10 AWS instance types.**

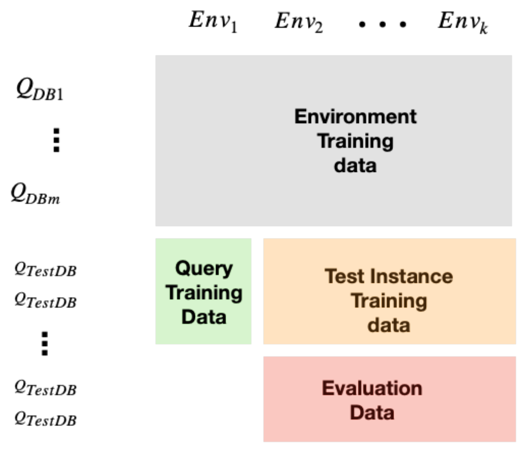

$Env_1$ $Env_2$ ... $Env_k$
$Q_{DB1}$
$Q_{DBm}$
$Q_{TestDB}$
Environment Training data
Query Training Data
Test Instance Training data
Evaluation Data

**Figure 5: Experiment data setup.**

of all instance types, thus eliminating a major source of performance fluctuation.

(1) I/O: We use magnetic disk, gp2 (networked ssd, but with variable speeds), and gp3 (more stable networked ssd) backends. Magnetic disks are generally the slowest.
(2) Memory: We use 4 GB, 8 GB, or 16 GB memory.
(3) CPUs: We use 2, and 4 cpu instance types.

Finally, we also include AWS Graviton, ARM and Intel based instance types. Using a combination of these properties, we select 10 instance types to run all the experiments. These are all among the smaller instance types. This is because our workloads are relatively small compared to modern cloud workload sizes as well — and it is only interesting to analyze cases where changes in the hardware cause distinct changes in performance. On large enough instances, all the workloads would exhibit almost no variance (similar to how there is little variance on the smaller workloads with the current systems in Table 2).

**System loads.** We consider either a single database queries running in isolation, or multiple database workloads running in different combinations of 2 – 6 at one time. The second scenario is based on multi-tenant systems such as Snowflake [60], or Salesforce [31].

## 6 EXPERIMENTS

We aim to train query prediction models which are specialized to particular database workloads, but robust to changes in hardware or system loads. In order to achieve this robustness, we use OS Pre-trained Transformer (Osprey), which learns how system states affect query runtime independent of the database workloads. We describe the experimental setup in Section 6.1, and show the key results in Section 6.2. We analyze why our model works in Section 6.3, and explore the effect of various design decisions for pre-training the Osprey model in Section 6.4.

### 6.1 Experimental Setup

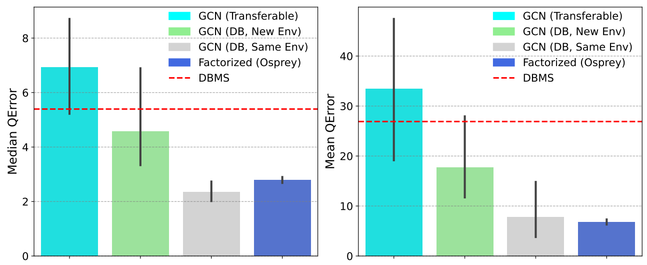
The experimental setup is based on the scenario where a workload is executed in one environmental context (hardware, system load), and we test it in a new environment. There are two factors to consider when defining a learned estimator: the model architecture, and the training data. In fact the data is even more important than the model architecture when considering generalization to new environments.

**Training / test data setup.** Our dataset can be viewed as a matrix where the rows are queries, and columns are different environments. An environment, $Env_i$, is defined by a <hardware, system load> pair, i.e., 10 AWS instance types, with system load being either single tenant (e.g., only IMDb queries) or specific multi-tenant scenarios (e.g., IMDb-TPCh-Stack queries). Figure 5 shows how we construct the training and test sets. For every experiment, there is a 'TestDB'. The 'environment training data' (gray) will include non TestDB samples on all environments. The 'query training data' (green) will include samples of TestDB queries on one environment — since it is unreasonable to expect having query training data of a new database on several different environments. The 'evaluation data' (red) will include new TestB queries on new environments. Finally, the 'test instance training data' is an idealized training set: it contains the training queries from TestDB, on all the test environments.

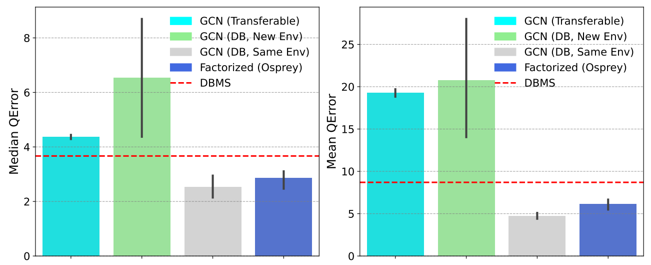
In our experiments, we fix all samples of 20% of the query workload from six AWS types (`a1_large_gp3`, `r7g_large_gp2`, `t3a_medium_gp3`, `m6a_large_mag`, `c7g_large_mag`, `t3_large_gp2`) as the evaluation data. For e.g., with 320 IMDb queries, we will exclude 64 test queries from the training set. The 6 instance types are chosen such that the workload execution times were distinctly different on each; thus it represents a challenging test set for generalization. All models are evaluated on the same samples, but are different in terms of the architecture and the training data. Fixing the test set allows us to more easily compare performance of learned models on different training data sources. As a running example to describe the training set, consider the test set: 'IMDb test queries, r7g_large_gp2 instance type, single tenant load'.

**DBMS Baseline.** Linear model (slope and coefficient) that takes as input the total DBMS query cost, and estimates latency, as used in several other papers [23, 49, 63]. We fit a separate line for each hardware type as in [63] — in our experiments, this does strictly better than one single linear model over all samples.

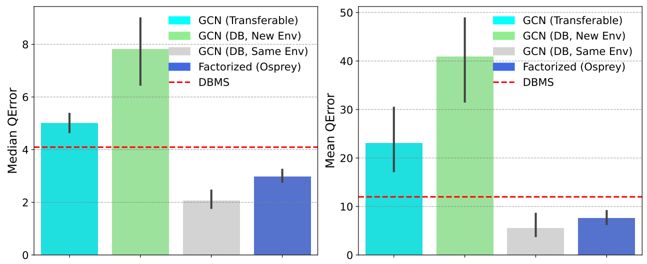
**GCN models.** There have been several proposed latency prediction neural net architectures that take as input query plan trees, such as



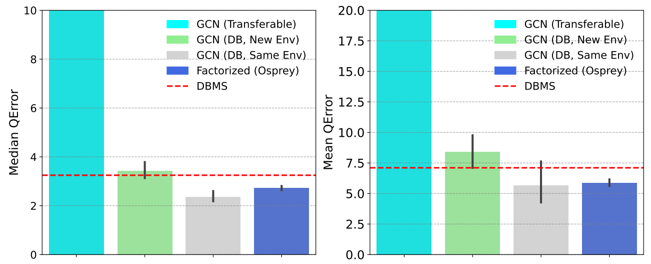
(a) IMDb.



(b) Stack.



(c) IMDb (Multi-tenant).



(d) Stack (Multi-Tenant).

**Figure 6: IMDb and Stack workload results.**

Tree Convolution Neural Networks (TCNN) [38, 39], or Graphical Convolutional Networks (GCN) [8, 22, 23, 53]. When evaluating on generalization to new system environments, we find all these architectures perform similarly, and we take GCN as the representative query plan model. There are two variants based on training data proposed in the literature: (1) GCN (Transferable) [23], which learns a single model over several database workloads. In particular, this model would use statistics based features instead of database specific features such as table names. In our example, GCN (Transferable) will be trained on all non-IMDb queries on 'r7g_large_gp2'. (2) GCN (DB), which is 'instance specialized' to a particular database workload, as is typically the case in proposed latency prediction models [8, 38, 39, 53, 54]. The benefit of such a model is that it can utilize database specific features, and learn correlations and other patterns particular to a workload; however, the drawback is that it can't use training data from other databases, and it may not be possible to have training data on different environments. Therefore, we consider two variants: (a) GCN (DB), Same Env: which is trained and tested on the same environment (e.g., trained on IMDb queries on 'r7g_large_gp2'). This is the ideal testing setup for such learned models, and (b) GCN (DB), New Env: which is trained on an a different environment (e.g., trained on IMDb queries on one of the non-test AWS instances).

**Factorized (Osprey).** Factorized model using Osprey as described in (§4). The Osprey module is pre-trained on 10% of the environment training data and fixed. This same pre-trained model is used in all evaluations (single and multi-tenant) of the test database. Then, only the DB specialized GCN module is trained on the query training data — this will have the same architecture and training data as GCN(DB, New Env) described above, e.g., trained on IMDb queries on one of the non-test AWS instances.

**Evaluation workloads.** To evaluate, we select the workloads from our dataset that show high variance across environments (see Table 2 in §5), and with sufficient number of queries so generalization to similar new queries is possible. This includes the workloads IMDb, Stack, and several of the workloads from RelationFit [23].

**Evaluation function.** Every sample has a latency, $y$, and each estimator predicts $\hat{y}$. There are several potential evaluation functions to compare these. For simplicity, we use Q-Error, or multiplicative error, as used by [23, 27]:

$$\text{Q-Error}(y, \hat{y}) = max(\frac{y}{\hat{y}}, \frac{\hat{y}}{y}) \tag{3}$$

The benefit of Q-Error is that it is symmetric, and easy to interpret. We also see very similar performance trends with relative error as used by [52, 63], but since it isn't symmetric, it leads to cases where over-estimates are penalized more, thus environments with lower runtimes tend to have higher errors.

**Loss function.** For training, we optimize for Q-Error as well. Since it is non-differentiable, we predict $log$(latencies), and optimize for the mean square error of these estimates, which is equivalent to optimizing for Q-Error as shown in [18].

## 6.2 Generalizing to new environments

Figure 6 shows the median and mean Q-Error of all the models on the test sets of IMDb and Stack in single or multi-tenant scenarios. We will go over the key trends that are reflected across all these results. The test set is fixed as described in Section 6.1, and the error bars represent each experiment with a different training environment — e.g., for the Factorized and GCN(DB, New Env) models: 4 experiments with the non-test hardware types in single-tenant, and 10 experiments with random environments in the multi-tenant scenario.

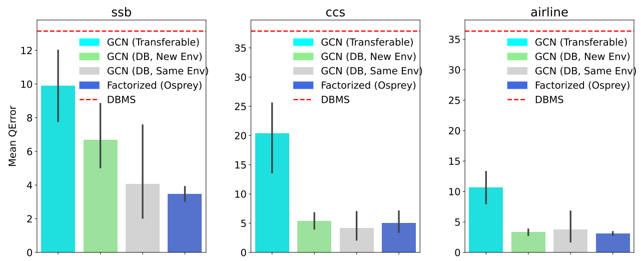
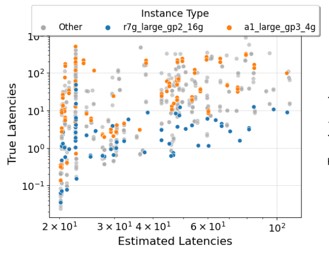
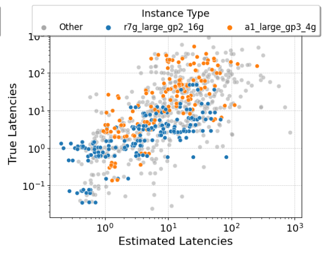


**Figure 7: Mean Q-Error across some of the smaller (1GB) Relation Fit workloads.**



**(a) Linear model fit to DBMS costs.**



**(b) Our model.**

**Figure 8: Scatterplot of estimated vs true latencies of the test IMDb queries across the test instance types.**

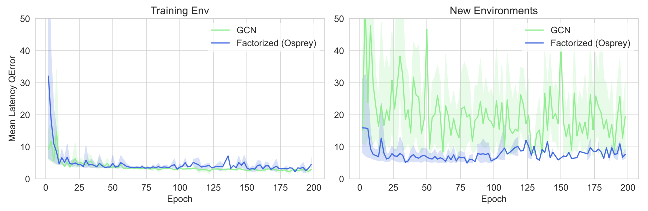


**Figure 9: Learning curves of the GCN model and Factorized (Osprey) model on the IMDb multi-tenant scenario. Each model is trained on IMDb queries from one environment (10 experiments with different training environments). Left is Q-Error on training environment, and right is on new environments.**

**GCN (Transferable).** This model struggles to generalize effectively to IMDb or Stack databases when trained on other databases in the same environment, performing worse than a simple DBMS baseline. Our evaluation scenario, compared to [23], presented unique challenges: (1) we had less training data, (2) the queries were more complex, and (3) crucially, the DB statistics varied greatly across workloads due to significant differences in DB sizes.

**Generalization gap between GCN (DB)'s.** GCN (DB, Same Env) shows the potential benefits of a DB specialized model — being up to 4× better at mean Q-Error than the DBMS baseline (in IMDb). However, this model needs training data of the Test DB on the test environment; when this model was trained on a different environment — GCN (DB, New Env), its performance is significantly worse than the DBMS baseline. This makes sense — as we saw from Table 2, there is significant variation among query latencies across hardware on IMDb and Stack, and multi-tenant scenarios add even more variance.

**Factorized (Osprey) model tested on new environments matches performance of DB and environment specialized GCN.** Factorized (Osprey) achieves similar performance as the GCN(DB, Same Env) model by specializing to the TestDB with a GCN model, and utilizing its knowledge about the new environments from the pre-trained Osprey model. We saw that GCN models trained alone on the TestDB or TestEnv were not enough to get good performance — thus, the Osprey model plays a critical role here. It essentially lets a GCN model trained on one environment to achieve the expected training environment performance on all new test environments, as if it had been trained specifically for the new environments. In fact, it is more robust than GCN, and shows less variance than the GCN(DB, Same Env) model. This is because even within the same environment, there can be variation in the system state which can't be captured by the query plan features alone, while a well calibrate system model can utilize these signals.

**RelationFit DBs.** Figure 7 shows the mean Q-Error across three of the smaller DBs (single tenant scenario). The smaller size of these DBs means there is less variance across environments since they can fit in RAM. Thus, there is not much of a generalization gap between the GCN(DB, New Env) and GCN(DB, Same Env) models. The Factorized (Osprey) model gets the same performance, but does slightly better on 'ssb', which is the RelationFit DB with highest variance across instances (see Table 2).

**Scatterplot.** Figure 8 shows a scatterplot of estimated vs true latencies of all IMDb test instances. Note the vertical lines of the DBMS cost estimator: this shows situations where similar costs have very different runtimes, thus a linear model using just these costs would have large error. Meanwhile, the predictions from the Factorized (Osprey) model roughly captures the true latency trend.

**Resource Usage.** Factorized model training consists of two parts. Pre-training the Osprey and latency prediction head is a one time cost; it takes around 5 hours on a V100 GPU, and roughly 5× longer on CPUs. Our pre-training data was about $15-20K$ samples depending on which TestDB we exclude. Training the instance specialized GCN model on the TestDB takes under 2 minutes, since its a lightweight model and just contains training data on one database and environment. CPUs are fast enough for inference on single samples. It takes $5ms$ for the GCN and prediction head inference, and about $25ms$ for Osprey. However, as we discussed in Section 4.3, the embeddings generated by Osprey can be amortized since they do not depend on the query, thus they do not add to the compile time costs.

## 6.3 Analysis and Other Use Cases

In this section, we dive deep into a single scenario — the IMDb workload with the multi-tenant load — to analyze what Osprey is learning and explore some qualitative use cases of Osprey. Specifically, we look at two system loads, with simultaneous execution of the workloads: IMDb-TPCh-Stack and IMDb-TPCh-Ergast. Recall,

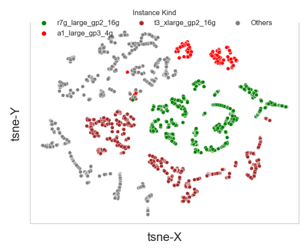


**Figure 10: t-SNE projection of unprocessed system logs.**

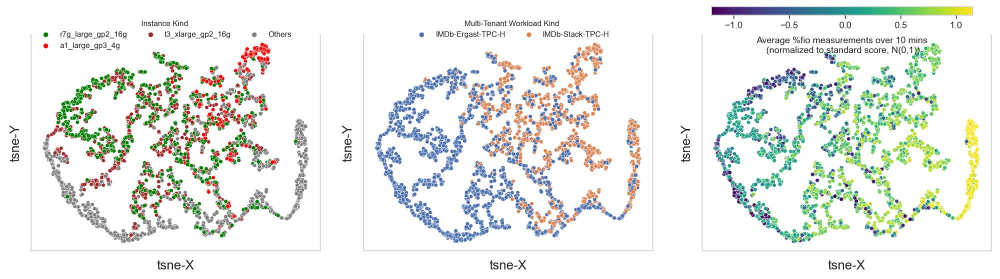


**Figure 11: t-SNE projection of the system embedding generated by OS Pre-trained Transformer. Each plot has the same points, colored to show different properties.**

from Table 2, Stack is a very large database, while Ergast is one of the 1$GB$ RelationFit DBs, thus we can expect considerably greater load on the environment with Stack.

**Learning curves.** Figure 9 shows the learning curves of our model and a GCN model on 10 experiments, each trained on one environment. In the training environment, both the models do very well. However, on new environments, the OS Pre-trained Transformer model consistently continues to improve predictions across all environments despite only training on one environment.

Both the models train using the same GCN architecture and training data. For a given query, our model will produce the same query vector with the GCN model in each environment; this will be combined with the embedding of the system state (generated by the pre-trained Osprey model) into different latency predictions. Therefore, we can look at these embeddings to see the signals the model may utilize to adapt its predictions to different environments.

**Visualizing system states.** We will use t-SNE (t-distributed Stochastic Neighbor Embedding) [56] to visualize the system states. t-SNE is an algorithm for dimensionality reduction to visualize relationships in high-dimensional datasets. First, let us consider the 2$d$ t-SNE projections of the unprocessed system logs. Figure 10 plots all samples in these IMDb multi-tenant workloads colored by AWS instance type. We see each hardware type get clustered together into separate islands; this makes intuitive sense because the scale of various measurements, such as kbmemfree, are probably most similar in the same instance type.

Next, we look at the same system logs, but pass them through Osprey to get an embedding for the system state. This is then projected to 2 dimensions using t-SNE. In Figure 11, we take the same set of 2$d$ projections, and visualize each data point using different criteria. First, we see the embeddings colored by instance types. We no longer see distinct islands for each instance type, instead, all the samples seem to be in a more continuous and mixed space.

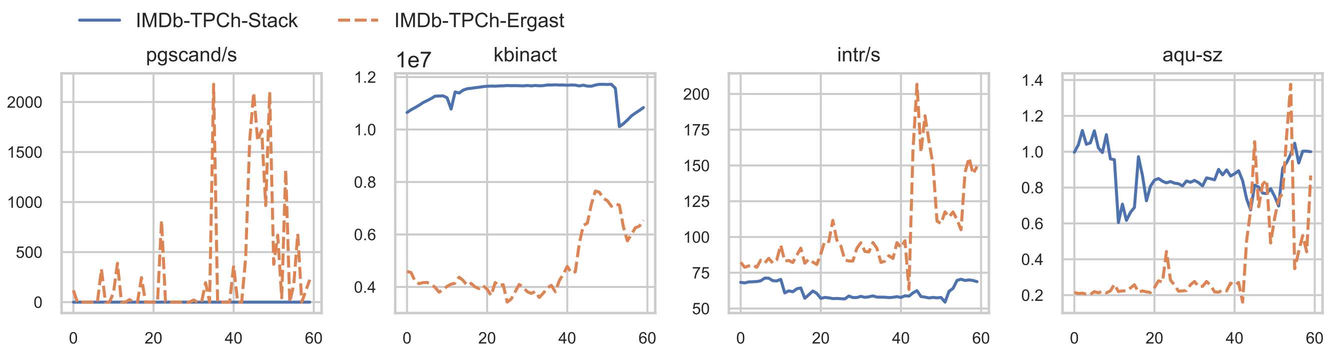
**Clustering workloads.** Next, we color the same t-SNE projections using the type of systeam load: IMDb-TPCh-Stack v/s IMDB-TPCh-Ergast. We see much clearer and distinct patterns here, in fact, each load is cleanly separated into two distinct groups. From a OS state embedding point of view, this is a very nice property, since we know that these loads have very different effects on the final performance. And it shows that on even very different hardware, there is a similarity in the type of effects created by a workload on the system.

**Clustering by performance properties.** Next, we color the points using (normalized) values for one of the I/O log measurements (%fio), resulting in a more continuous spectrum (based on the system measurements) across the samples. This implies that Osprey assimilates the system measurements and positions them in a new embedding space where elements are arrayed along a continuum of factors that influence performance.

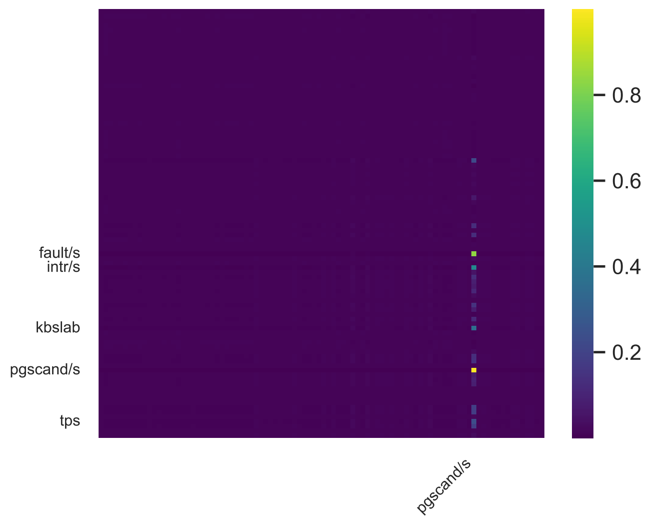
The previous two clustering results suggest that Osprey embeddings could be useful in finding similar workload patterns, or classifying workload patterns — similar to how natural language embeddings enable searching for similar documents based on semantic meaning [47], such approaches may make it possible to embed workloads using Osprey and then search to find ones that have similar performance properties across different hardware scales.

**Single query example.** Zooming in further, we analyze a single example query, 5a104.sql, on the instance type r7g_large_gp2. It takes 1 second with the IMDb-TPCh-Ergast load and over 400 seconds with the IMDb-TPCh-Stack system load. This is likely due to the query running entirely from memory v/s requiring extensive disk access and stalling due to the pressure of the Stack workload. From Osprey's perspective, it encounters 92 tokens of recent measurements, such as the four shown in Figure 12a. Notably, for IMDb-TPCh-Ergast, we observe significantly higher values of `pgscand/s`, which indicates pages scanned per second from page tables. This suggests that the queries are being executed from memory, contributing to faster performance. The model then applies the transformer layers to these inputs. To understand why Osprey embeds these two samples very differently, we will need to look at the internals of the transformer's attention module.

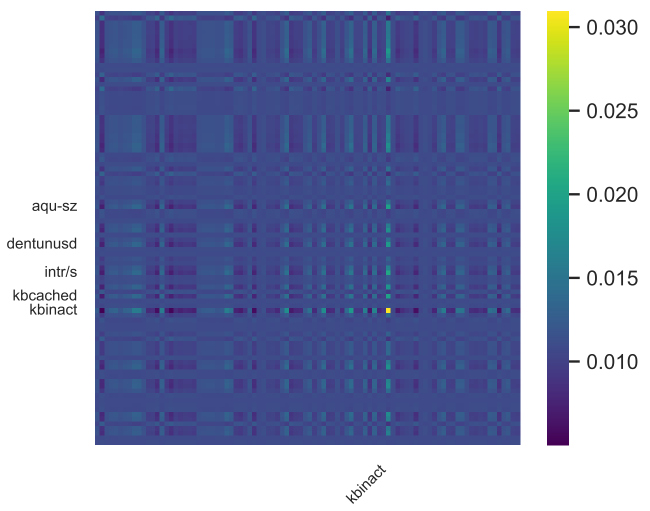
**Attention scores.** In Figure 12, we examine the attention scores assigned to various system log measurements for these inputs (refer to §4.2 for a description of attention scores). In the first attention matrix, we see that `pgscand/s` has very high attention scores for several rows (i.e., the $a_{ij}$ values for $j$ corresponding to `pgscand/s` are much higher for several different $i$'s). This means the $Emb_i$ embedding outputs from this layer would be disproportionately influenced by the `pgscand/s` log measurements. This is very different from the attention scores on the IMDb-TPCh-Stack workload, which are fairly evenly spread out through different logs — thus



**(a) System logs in the last 10 mins.**



**(b) IMDb-TPCh-Ergast.**



**(c) IMDb-TPCh-Stack.**

**Figure 12: Query 5a850.sql (IMDb) on r7g_large_gp2_16g under two different system loads: (1) IMDb-TPCh-Ergast (taking 1 second) (2) IMDb-TPCh-Stack (taking 400 seconds). First row shows some of the system logs for previous 10 minutes (the input to Osprey), and row 2 shows an attention matrix in the first layer of Osprey in both scenarios.**

the embedding outputs would be more balanced. From the perspective of our prediction task, in the IMDb-TPCh-Ergast, the model reweighs the inputs such that the `pgscand/s` log is much more influential — and intuitively, this makes sense, as this log signifies that the queries on that system are utilizing the page table system memory very well.

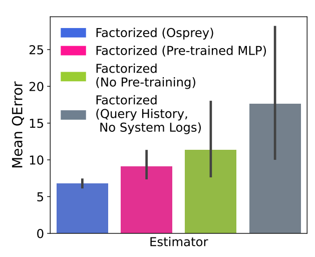
**Attention scores as proxies for most relevant metrics.** These attention scores essentially highlighted the 1 of 92 measurements most critical to explain the better performance in the IMDb-TPCh-Ergast scenario. In this case, we had strong intuition for this, because the workload patterns were constructed to be understood easily by us, but in general, this could be a useful debugging tool in more complex system environments with even more logging measurements.

## 6.4 Pre-training Design

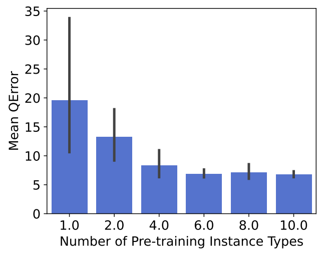
In this section, we will address various design decisions about the Osprey model, and the pre-training data requirements.

*6.4.1 Architecture choices.* Figure 13a compares results of Factorized (Osprey) with some other interesting design choices. We will discuss each in turn.

**Was a transformer necessary?** With Factorized (Pretrained MLP), we could approach the performance of Osprey by utilizing the same factorized pre-training setup. The performance gap may be attributed to the benefits of attention analyzed in Section 6.3. Moreover, MLPs are not flexible, for e.g., if we wanted to add per-cpu or per-I/O logs, since MLPs require fixed dimensions. Nevertheless, in some situations, using Pretrained MLPs could be useful because its a simpler model, and faster for both training and inference.



**(a) Different versions of the factorized architecture.**



**(b) Varying number of instance types in pre-training data.**

**Figure 13: Experiments exploring pre-training design.**

**Was pre-training necessary?** Factorized (No Pre-training) model uses the same architecture as Factorized (Osprey) but the transformer is not pre-trained; instead, it trains from scratch on the pre-training + IMDb's queries on one environment. This isn't practical because it doesn't support adding new databases, and adds hours to each training run (v/s a one time pre-training cost of Osprey), but here we analyze it to better understand why Osprey helps. Osprey's clearly better performance seems to be because without pre-training, the GCN is fitting predictions to all the databases in the workload, which is much harder than specializing to one database. In comparison, the Osprey model first learns about how the environment state affects latencies and then fixes this model (in pre-training). Then, it adapts a lightweight GCN model to just one database, separating the learning of the DB and environment.

**Can we use query history instead of system logs?** System logs may not always be available, therefore, it is natural to ask if we can use the recent query history as a proxy for system logs. If there was only query in the workload, this would make perfect sense, as the recent runtimes would exactly tell us how the system state would affect the latency of the next runs. In order to test this, we flatten the estimated and actual query plan features (cost and actual runtime, estimated and actual cardinality etc.) into a vector, and a sequence of recent vectors becomes an input to the transformer model (instead of system logs). However, this does a lot worse than Osprey. We believe this is because:(1) this approach mixes the DB representation and system representation, and this leads to information leakage between the system and query representation, for e.g.,, the model may learn to do well in pre-training by learning about queries instead of the system. (2) query latencies vary widely — for e.g., small queries can run fast even in very overloaded systems.

*6.4.2 Pre-training data.* Figure 13b answers the question, do we need to see every possible hardware instance type in pre-training? There are hundreds of potential AWS instance types, so this is important. We pre-train Osprey with 1,2,4,6,8, or all the 10 instance types and test the generalization performance on IMDb (single-tenant) workloads. Pre-training data having 6 instance types, or more, seems to do as good as having all instances; this is because a lot of instance types have overlap, since different hardware configurations are usually a mix and match of a few basic blocks. Therefore,it is natural to expect that we should not need to see every possible combination as long as we see enough coverage of individual components.

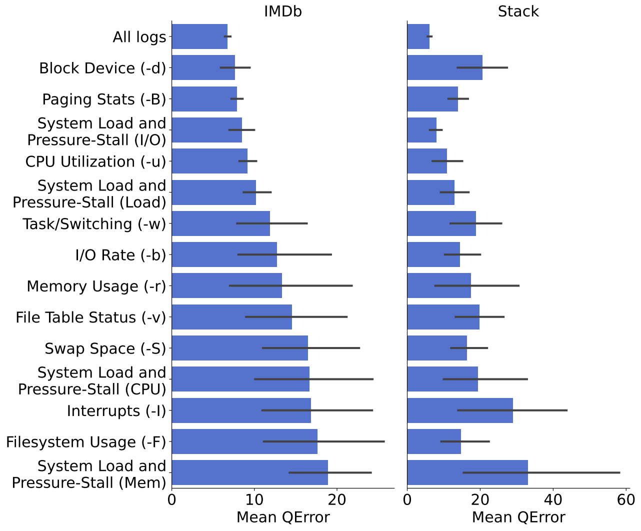


**Figure 14: Ablation study on using just one of the different SAR Linux logs.**

*6.4.3 Log Files.* We used several different log files from Linux's SAR utilities to define the environment state (see Table 1). In Figure 14, we use only one file at a time to define the environment, and pre-train appropriate Osprey models for each log file, and evaluate on single tenant IMDb and Stack workloads. This gives us a sense of relative importance of log files to the final performance as well. On IMDb, we find that 'Block Device', 'Paging', or 'I/O Pressure Stall' logs all give almost optimal performance; this is probably because the most important fact in IMDb workloads is the RAM state, and how much I/O is being done in the system. However, on the Stack workload, no single log file gives performance better than a baseline — suggesting more complicated explanations, since the Stack workload is larger and doesn't fit into memory.

## 7 CONCLUSION

We introduced Osprey, which maps linux system logs to an embedding space useful for downstream tasks like predicting query latencies or clustering different workloads based on their performance characteristics. Our evaluation is a proof of concept on a restricted scenario with only 10 AWS instance types, but the overall recipe of pre-training models that take into account OS states can be extended to a much larger scale.

In the future, we plan to explore OS state embeddings as a building block for other downstream systems tasks where ML has shown promise, such as compilers [33] or scheduling jobs [37]. These tasks are also affected by OS states, which suggests a potential to apply the techniques behind Osprey more broadly. For instance, we could replace the query model with a model that takes as input featurization of other arbitrary tasks, like compiler graphs, and predicts their performance.

## REFERENCES

[1] 2012. Systat Utilities. https://sysstat.github.io/ [Online;].

[2] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based query performance modeling and prediction. In 2012 IEEE 28th International Conference on Data Engineering. IEEE, 390–401.

[3] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, spring joint computer conference. 483–485.

[4] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. Proceedings of the VLDB Endowment 16, 12 (2023), 3515–3527.

[5] Peter Bloem. 2019. Transformers from scratch. VU University: Amsterdam, The Netherlands (2019).

[6] Christian Böhm. 2000. A cost model for query processing in high dimensional data spaces. ACM Transactions on Database Systems (TODS) 25, 2 (2000), 129–178.

[7] Surajit Chaudhuri and Kyuseok Shim. 1995. An overview of cost-based optimization of queries with aggregates. IEEE Data Eng. Bull. 18, 3 (1995), 3–9.

[8] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Join Order Selection Learning with Graph-based Representation. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. 97–107.

[9] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. Leon: a new framework for ml-aided query optimization. Proceedings of the VLDB Endowment 16, 9 (2023), 2261–2273.

[10] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In Slides from Linux Kongress, Vol. 18. 1–42.

[11] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. ACM SIGPLAN Notices 48, 4 (2013), 77–88.

[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. ACM SIGPLAN Notices 49, 4 (2014), 127–144.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).

[14] Jialin Ding, Ryan Marcus, Andreas Kipf, Vikram Nathan, Aniruddha Nrusimha, Kapil Vaidya, Alexander van Renen, and Tim Kraska. 2022. SageDB: An Instance-Optimized Data Analytics System. Proceedings of the VLDB Endowment 15, 13 (2022), 4062–4078.

[15] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. 1992. Query optimization in a heterogeneous dbms. In VLDB, Vol. 92. 277–291.

[16] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. 337–348.

[17] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the impact of memory allocation on high-performance query processing. In Proceedings of the 15th International Workshop on Data Management on New Hardware. 1–3.

[18] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. Proc. VLDB Endow. 12, 9 (2019), 1044–1057. https://doi.org/10.14778/3329772.3329780

[19] Zhengjie Gao, Ao Feng, Xinyu Song, and Xi Wu. 2019. Target-dependent sentiment classification with BERT. Ieee Access 7 (2019), 154290–154299.

[20] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In Proceedings of IEEE 9th international conference on data engineering. IEEE, 209–218.

[21] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. arXiv preprint arXiv:2109.05877 (2021).

[22] Roman Heinrich, Manisha Luthra, Harald Kornmayer, and Carsten Binnig. 2022. Zero-shot cost models for distributed stream processing. In Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems. 85–90.

[23] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-shot cost models for out-of-the-box learned cost prediction. arXiv preprint arXiv:2201.00561 (2022).

[24] Windsor W Hsu and Alan Jay Smith. 2004. The performance impact of I/O optimizations and disk improvements. IBM Journal of Research and Development 48, 2 (2004), 255–289.

[25] Yannis E Ioannidis. 1996. Query optimization. ACM Computing Surveys (CSUR) 28, 1 (1996), 121–123.

[26] Zhenlan Ji, Pingchuan Ma, and Shuai Wang. 2022. PerfCE: Performance debugging on databases with chaos engineering-enhanced causality analysis. arXiv preprint arXiv:2207.08369 (2022).

[27] Yuri Kim, Yewon Choi, Yujung Gil, Sanghee Lee, Heesik Shin, and Jaehyok Chong. 2023. BitE: Accelerating Learned Query Optimization in a Mixed-Workload Environment. arXiv preprint arXiv:2306.00845 (2023).

[28] Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating filtered group-by queries is hard: Deep learning to the rescue. In 1st International Workshop on Applied AI for Database Systems and Applications.

[29] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016).

[30] Virginia Klema and Alan Laub. 1980. The singular value decomposition: Its computation and some applications. IEEE Transactions on automatic control 25, 2 (1980), 164–176.

[31] Heiko Koziolek. 2011. The sposad architectural style for multi-tenant software applications. In 2011 Ninth Working IEEE/IFIP Conference on Software Architecture. IEEE, 320–327.

[32] 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. 489–504.

[33] Hugh Leather and Chris Cummins. 2020. Machine learning in compilers: Past, present and future. In 2020 Forum for Specification and Design Languages (FDL). IEEE, 1–8.

[34] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? Proc. VLDB Endow. 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[35] Yao Lu, Srikanth Kandula, Arnd Christian König, and Surajit Chaudhuri. 2021. Pre-training summarization models of structured datasets for cardinality estimation. Proceedings of the VLDB Endowment 15, 3 (2021), 414–426.

[36] Michael V Mannino, Paicheng Chu, and Thomas Sager. 1988. Statistical profile estimation in database systems. ACM Computing Surveys (CSUR) 20, 3 (1988), 191–221.

[37] 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.

[38] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making learned query optimization practical. ACM SIGMOD Record 51, 1 (2022), 6–13.

[39] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. arXiv preprint arXiv:1904.03711 (2019).

[40] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management. 1–4.

[41] Vivek Narasayya and Surajit Chaudhuri. 2022. Multi-Tenant Cloud Data Services: State-of-the-Art, Challenges and Opportunities. In Proceedings of the 2022 International Conference on Management of Data. 2465–2473.

[42] 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 Proceedings of the 2021 International Conference on Management of Data. 2557–2569.

[43] Parimarjan Negi, Ryan C. Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. Proc. VLDB Endow. 14, 11 (2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[44] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. Proceedings of the VLDB Endowment 16, 6 (2023), 1520–1533.

[45] Andreas Kipf Hongzi Mao Nesime Tatbul Tim Kraska Mohammad Alizadeh Parimarjan Negi, Ryan Marcus. 2021. Cardinality Estimation Benchmark. https://github.com/learnedsystems/ceb [Online;].

[46] Nicolas Poggi, Víctor Cuevas-Vicenttín, Josep Lluis Berral, Thomas Fenech, Gonzalo Gómez, Davide Brini, Alejandro Montero, David Carrera, Umar Farooq Minhas, Jose A Blakeley, et al. 2020. Benchmarking Elastic Cloud Big Data Services under SLA Constraints. In Performance Evaluation and Benchmarking for the Era of Cloud (s) 11th TPC Technology Conference, TPCTC 2019, Los Angeles, CA, USA, August 26, 2019, Revised Selected Papers 11. Springer, 1–18.

[47] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. OpenAI blog 1, 8 (2019), 9.

[48] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 10684–10695.

[49] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan Narayanaswamy. 2023. Auto-WLM: Machine learning enhanced workload management in Amazon Redshift. In Companion of the 2023

International Conference on Management of Data. 225–237.

[50] Patricia G Selinger. 1990. The impact of hardware on database systems. In IBM Germany Scientific Symposium Series. Springer, 316–334.

[51] Praveen Seshadri, Joseph M Hellerstein, Hamid Pirahesh, TY Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J Stuckey, and S Sudarshan. 1996. Cost-based optimization for magic: Algebra and implementation. In Proceedings of the 1996 ACM SIGMOD international conference on Management of data. 435–446.

[52] 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 ACM SIGMOD International Conference on Management of Data. 99–113.

[53] Yuanfeng Song, Yuqiang Li, Shuhuan Fan, Dongsheng He, and Jianming Liao. 2022. A New Graph Neural Network-based Join Optimization Algorithm. In 2022 International Conference on Algorithms, Data Mining, and Information Technology (ADMIT). IEEE, 20–24.

[54] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. arXiv preprint arXiv:1906.02560 (2019).

[55] Yannis Theodoridis, Emmanuel Stefanakis, and Timos Sellis. 1998. Cost models for join queries in spatial databases. In Proceedings 14th International Conference on Data Engineering. IEEE, 476–483.

[56] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. Journal of machine learning research 9, 11 (2008).

[57] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. Proceedings of the VLDB Endowment 16, 6 (2023), 1413–1425.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. Advances in neural information processing systems 30 (2017).

[59] Francesco Ventura, Zoi Kaoudi, Jorge Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Expand your training limits! generating training data for ml-based data management. In Proceedings of the 2021 International Conference on Management of Data. 1865–1878.

[60] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). USENIX Association, Santa Clara, CA, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[61] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. Proceedings of the VLDB Endowment 12, 3 (2018), 210–222.

[62] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F Naughton. 2013. Towards predicting query execution time for concurrent and dynamic database workloads. Proceedings of the VLDB Endowment 6, 10 (2013), 925–936.

[63] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In 2013 IEEE 29th International Conference on Data Engineering (ICDE). IEEE, 1081–1092.

[64] Ziniu Wu, Pei Yu, Peilun Yang, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. 2021. A unified transferable model for ml-enhanced dbms. arXiv preprint arXiv:2105.02418 (2021).

[65] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In Proceedings of the 2022 International Conference on Management of Data. 931–944.

[66] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a steered query optimizer in production at Microsoft. In Proceedings of the 2022 International Conference on Management of Data. 2299–2311.

[67] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: a tree transformer model for query plan representation. Proceedings of the VLDB Endowment 15, 8 (2022), 1658–1670.

[68] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query performance prediction for concurrent queries using graph embedding. Proceedings of the VLDB Endowment 13, 9 (2020), 1416–1428.

[69] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. arXiv preprint arXiv:2302.06873 (2023).