Auto-WLM: machine learning enhanced workload management in Amazon Redshift

Gaurav Saxena
gssaxena@amazon.com
Amazon Web Services
USA

Mohammad Rahman
rerahman@amazon.com
Amazon Web Services
USA

Naresh Chainani
nareshkc@amazon.com
Amazon Web Services
USA

Chunbin Lin*
chulin@visa.com
VISA
USA

George Caragea†
george.caragea@lacework.net
Lacework
USA

Fahim Chowdhury
fahimtc@amazon.com
Amazon Web Services
USA

Ryan Marcus
rcmarcus@seas.upenn.edu
Amazon Web Services
University of Pennsylvania
USA

Tim Kraska
kraska@mit.edu
Amazon Web Services
MIT
USA

Ippokratis Pandis
ippo@amazon.com
Amazon Web Services
USA

Balakrishnan (Murali) Narayanaswamy
muralibn@amazon.com
Amazon Web Services
USA

ABSTRACT

There has been a lot of excitement around using machine learning to improve the performance and usability of database systems. However, few of these techniques have actually been used in the critical path of customer-facing database services. In this paper, we describe Auto-WLM, a machine learning based automatic workload manager currently used in production in Amazon Redshift. Auto-WLM is an example of how machine learning can improve the performance of large data-warehouses in practice and at scale. Auto-WLM intelligently schedules workloads to maximize throughput and horizontally scales clusters in response to workload spikes. While traditional heuristic-based workload management requires a lot of manual tuning (e.g. of the concurrency level, memory allocated to queries etc.) for each specific workload, Auto-WLM does this tuning automatically and as a result is able to quickly adapt and react to workload changes and demand spikes. At its core, Auto-WLM uses locally-trained query performance models to predict the query execution time and memory needs for each query, and uses this to make intelligent scheduling decisions. Currently, Auto-WLM makes millions of decisions every day, and constantly optimizes the performance for each individual Amazon Redshift cluster. In this paper, we will describe the advantages and challenges of implementing and deploying Auto-WLM, as well as outline areas of research that may be of interest to those in the "ML for systems" community with an eye for practicality.

ACM Reference Format:
Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan (Murali) Narayanaswamy. 2023. Auto-WLM: machine learning enhanced workload management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data (SIGMOD-Companion '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3555041.3589677

1 INTRODUCTION

Amazon Redshift uses machine learning in a number of different ways. For example, Redshift uses models to decide when to automatically create materialized views, trigger "vacuums" of tables, to recommend and create and sort-keys, and more [2]. These optimizations are largely high-level (e.g., when to create a materialized view) and knob-tuning as they adjust traditional database components and have been explored in other work [11, 50]. More recently ML-enhanced, or instance-optimized, components [20, 23, 26–28, 43, 54, 55] have been proposed, which go beyond the tuning of traditional components and deeply embed machine learning models into the database. However, so far the adoption of ML-enhanced components has been limited despite the potential performance advantages, as reliably integrating machine learning into the *critical path* of any database management system is far from trivial.

*Work performed while at AWS
†Work performed while at AWS

SIGMOD-Companion '23, June 18–23, 2023, Seattle, WA, USA

ACM ISBN 978-1-4503-9507-6/23/06.
https://doi.org/10.1145/3555041.3589677

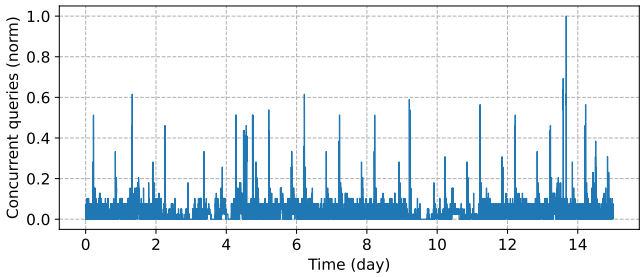

Figure 1: Concurrency changes on a real workload.

In this paper, we describe Auto-WLM, Amazon Redshift's in production, machine learning based automatic workload manager. Auto-WLM serves as an example of how ML-enhanced components can be made practical. As of today, Auto-WLM schedules every query on Amazon Redshift, using query run-time and memory prediction models. This includes decisions on the optimal number of concurrent queries (i.e., admission control), priority of queries (i.e., scheduling), and when to horizontally scale the cluster to increase its capacity (i.e., elasticity).

Cloud data-warehouses, like Amazon Redshift, are expected to efficiently run mixed workloads consisting of read and write, long and short, and local and federated queries. As different types of queries have very different resource requirements, tuning the query admission/scheduling policy is complex and even expert DB administrators may struggle with it. To make matters worse, workloads are rarely constant. For example, in the morning when analysts start their work, we can often observe a peak of short-running dashboard queries which normalizes throughout the day. ETL queries tend to run during the night, but can vary widely in complexity and size depending on the accumulated data. In addition, urgent ad-hoc business needs can put a lot of load on a system at unexpected times. Customers need their database to scale to changing workloads at a reasonable cost, and meeting these needs can be a substantial challenge for administrators.

On-premise database systems (e.g., Teradata, IBM DB2, SQL Server) typically implement workload isolation techniques, such as query queues. These longstanding techniques allow users to create multiple queues to divide the resources of a database system to control the amount of memory and CPU each query gets as well control the number of concurrent queries [1, 15]. This works as long as the workload is mostly static and homogeneous, but breaks down when the workload is dynamic, requiring constant adjustment. In addition, if the difference between the peak and the average workload is high, then the choice before customers is to either under-provision the database system and live with performance issues at peak time, or size for peak and overpay for rest of the workload.

To address this, modern cloud database systems offer automatic horizontal scaling, which dynamically adds additional compute nodes to handle peak query volumes, and then removes the additional nodes when they are no longer needed. Figure 1 shows the number of concurrent queries running on a Redshift cluster across 15 days. There are significant spikes in concurrency that need to be handled. Determining the optimal time to scale horizontally (i.e., add new nodes), increase concurrency, or change scheduling policies is a non-trivial task and requires understanding the resource demands of each query. For example, if horizontal scaling is done prematurely, resources are unnecessarily wasted, and if horizontal scaling is done too late, overall query performance is negatively impacted. Additionally, whenever queries queue there is a risk of disproportional increase in the response time of short running queries compared to their execution time; a phenomenon often referred to as head of the line blocking (HLB).

In this paper, we describe an autonomous workload management system, called Auto-WLM, which addresses all the use cases outlined above while minimizing the knobs customers have to turn. First, Auto-WLM uses machine learning models to estimate each query's memory and CPU requirements. This model is automatically retrained and instance optimized to the customer's cluster. Second, a queuing theory model based on predicted execution time of queries determines the right number of queries to execute concurrently, to achieve the highest throughput. Third, if some queries cannot be executed using currently available resources, Auto-WLM horizontally scales database resources to execute them. Fourth, short query acceleration — which prioritizes selected short-running queries ahead of longer-running queries — helps to mitigate HLB issues. Auto-WLM allows advanced users to further control the cost performance trade-off by setting query or query queue priorities. Finally, Auto-WLM allows users to specify query monitoring rules and automatically aborts queries which violate them, to protect against unnecessary abuse of resources.

While Auto-WLM is not the first ML-enhanced workload manager (or scheduler) for data systems [26, 29, 43, 46, 57], to the best of our knowledge Auto-WLM is the first used in a customer-facing production system, and that combines elasticity decisions with scheduling decisions for online workloads. Making Auto-WLM practical required several iterations and a careful design, in particular ensuring that there are no surprises when faced with the long-tail of workloads that Amazon Redshift serves. Moreover, as Auto-WLM is in the critical path of every query, it is of utmost importance that the time taken to make decisions does not add significant overhead, especially for short-running queries. We will conclude the paper with lessons learned from implementing Auto-WLM and research challenges we believe are currently not well addressed in the research field.

In summary, we make the following contributions:

- We describe Auto-WLM, a ML-enhanced workload manager used in Amazon Redshift.
- We show how Auto-WLM uses ML-models with a dynamic concurrency algorithm to maximize throughput, minimize latency, and make scaling decisions.
- We show comprehensive experiments on public benchmarks and production data to evaluate the performance of Auto-WLM.
- We summarize the challenges of implementing Auto-WLM and outline important research challenges, which we believe are not yet addressed by the research community.

2 RELATED WORK

Machine Learning for Databases: Recently, machine learning techniques have had a large impact on the administration and management of database systems. Machine learning systems have been used for index recommendation [8, 9], configuration tuning [11, 50],

semantic queries [4], entity matching [35], and workload management [29, 47]. In contrast, ML-enhanced components, which embed ML-models deep inside the database to better adjust to specific data or workloads, such as learned index structures [13, 21, 23], learned storage layouts [7, 10, 32, 36], learned scheduling algorithms, [26, 43, 57], or learned query optimization [27, 28, 54, 56]) — to the best of our knowledge — have so far not been adopted for *customer-facing* database services. Some success stories exists for internal workloads, where performance regressions are often less concerning than in customer-facing deployments: for example, the learned index integration into Google BigTable [17] and the Bao-like learned query steering in Microsoft's Scope [37, 58]. We believe Auto-WLM is the first ML-enhanced component deployed in a customer-facing data-warehouse service.

Workload Management in Databases: A database workload management component has to make three different types of decisions: admission, scheduling and execution resource control (elasticity). AutoWLM uses a combination of smart algorithms and ML models to perform all three. In each sub-area, significant prior work exists. Summarizing each is beyond the scope of this paper. In the following, we focus mainly on recent work in using machine-learning to improve admission, scheduling, and resource management.

In [52], a query scheduler was proposed for the Umbra research database to self-tune the hyper-parameters of its fixed scheduling policy for each input workload. In [40], an efficient query scheduler for micro-tasks was proposed with tuned implementations for many heuristic policies (e.g., fair, highest priority first, and proportional priority). Decima [26] uses RL to fully-learn a jobs scheduler on large clusters for Spark jobs. LSched [43] and [57] present novel RL-based scheduling algorithm for analytical workloads, and aim to take the state of the system and the intrinsics of query plans into account. [29, 30] give supervised and reinforcement learning solutions for scheduling and resource management, respectively. [18, 49] use reinforcement learning during the query execution process to adapt query plans based on live feedback. Admission control has also been addressed using learned approaches [42, 48].

While these research results show a lot of promise, they did not yet solve the practical issues we faced when developing Auto-WLM. For example, Decima and LSched both assume that the number of servers is fixed, whereas in Redshift it is possible to allocate additional nodes. Moreover, Auto-WLM combines admission control, scheduling, and resource control into one component, whereas many existing efforts focused on just scheduling. A notable exception is [29], which however deals only with batch workloads. However, as we show throughout this paper, these decisions are all highly intertwined and should be addressed together in order to create even a basic "self-driving" database [41]. For example, if we make a prediction about the memory resources needs of a query to make a scheduling decision, it is only reasonable to use the same prediction for resource control during query execution.

3 OVERVIEW

In this section, we describe the high level architecture of Amazon Redshift Auto-WLM. Each Amazon Redshift database is split into two parts: (1) the *main* cluster, which is always running whenever the database is up, and (2) the *concurrency scaling* clusters which are created and destroyed as needed to deal with spikes in the workload. Auto-WLM manages both cluster types, and contains the following five major components:

- **Query performance model (ML predictor)**: a machine learning model that is responsible for predicting the latency and memory requirements of a query.
- **Query prioritization strategy (Priority Assigner)**: a strategy to assign queries higher or lower priorities based on the predictions from the model (e.g., shortest job first).
- **Admission controller**: a module to determine whether or not an incoming query should wait in the queue, be executed on the user's "main" cluster, executed in a special "short" query queue, or sent to a concurrency scaling cluster.
- **Utilization monitor**: a module that computes the utilization of cluster resources, and informs the admission control module if more or fewer queries should be admitted.
- **ML trainer**: a module responsible for storing a sliding window of training data and periodically updating the query performance model, which must be done in a low-overhead way.

Figure 2 shows an overview of the architecture. To understand how Auto-WLM works, we will briefly walk through the life of a query on an Amazon Redshift cluster: when a query arrives, the ML Predictor predicts the needed memory, needed CPU time, and execution time for the query. Based on the predicted execution time, the Priority Assigner assigns the query a priority, with the goal of assigning high priorities to queries with low resource requirements and fast execution times. By executing such queries with high priority, Auto-WLM emulates an approximate shortest-job-first scheduling, which minimizes average query latency. Once a query has an assigned priority, the Admission Controller decides where the query will go. The Admission Controller, for each query, chooses one of:

(1) place the query into the short query queue (described in Section 4.2.1) if the query's estimated resource utilization is low and the query was assigned a high priority,
(2) execute the query on the user's main cluster, executing it when the predicated amount of memory and CPU time are available and doing so is not expected to cause a drop in throughput (described in Section 4.2.2),
(3) place the query on a concurrency scaling cluster, if an existing concurrency scaling cluster has sufficient resources,
(4) or, if none of the above options are available, place the query in the main cluster's queue with the assigned priority. If too many queries are queued, a new concurrency scaling cluster may be launched.

Once the query finishes executing, the Utilization Monitor computes the current system utilization and gives suggestion to the Admission Controller to admit more or less queries based on whether the system is under-utilized or over-utilized. In addition, the ML Trainer will re-train the models periodically in order to get better prediction quality. The next section discusses each of these components in detail.

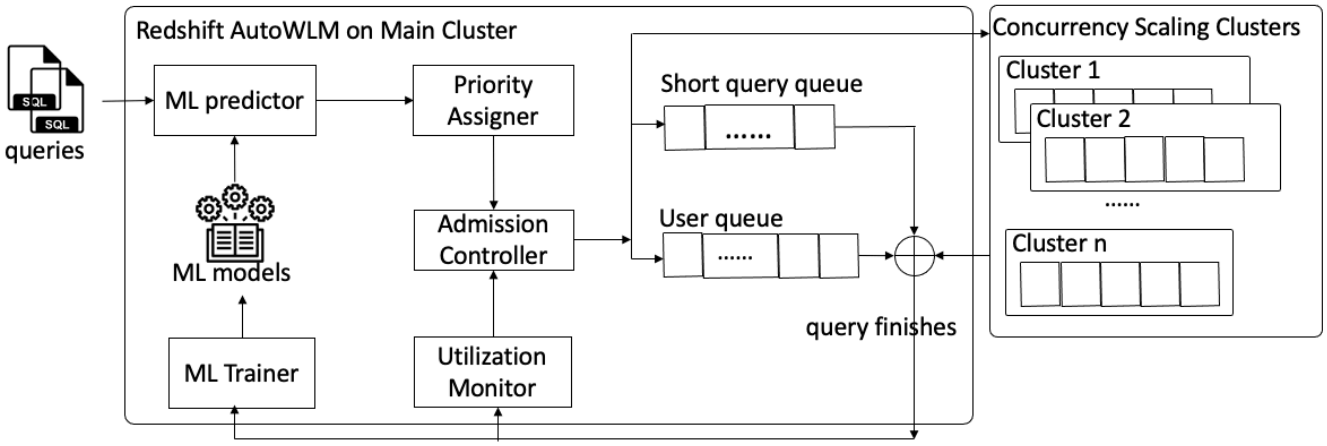

Figure 2: AutoWLM architecture overview. When a query arrives, the ML predictor determines the resource requirements of the query. The priority assigner determines how short the incoming query is to support shortest-job-first scheduling. Given the priority, the admission controller decides if the query should be executed on the main's clusters short query queue, user queue, or sent to a concurrency scaling cluster. Once a query finishes, two things happen: (1) the ML trainer adds the observed query plan and latency to the training set, and may update the ML predictor, and (2) the utilization monitor determines if the cluster's resources are under- or over-subscribed, telling the admission controller to let in more/fewer queries.

4 AUTOWLM

In this section, we describe each of Auto-WLM's modules in detail, discussing their design and implementation. Section 4.1 describes the ML predictor and ML trainer, focusing on the specific latency and overhead requirements on Amazon Redshift clusters. In Section 4.2, we describe the admission controller, and how Amazon Redshift automatically adapts the cluster's concurrency level to fit the user's needs, as well as how new concurrency scaling clusters are spun up and down. Finally, in Section 4.2.3, we explain how the Priority Assigner helps keep average latency low by considering both a query's potential execution time and resource requirements.

4.1 Machine learning models in Auto-WLM

All of Auto-WLM's intelligence depends on the accuracy of a solid query execution and resource model. If we can accurately predict the time and resource requirements of a query, then scheduling each query correctly becomes significantly easier.

Requirements. Unlike prior work on query performance modeling [12, 14, 31, 33, 51], Auto-WLM's query model has several unique requirements. First, Auto-WLM's query model is trained locally on the production database cluster, so training overhead needs to be low in order to avoid interference with query processing. This local training also simplifies legal constraints, as the user's data never leaves their own cluster. Second, many Amazon Redshift queries execute in just a few milliseconds, so any query model thus needs inference times at the microsecond level to avoid a significant increase in the runtime of small queries. Third, Auto-WLM's query model needs to be instance-optimized, that is, custom-tailored to the user's workload. A pre-trained model that worked for 95% of customers but failed for 5% of customers would not be acceptable. In fact, we found that Auto-WLM's simple, instance-optimized "local" models outperformed more sophisticated models that were not instance-optimized (see Section 6 for more discussion).

To satisfy these three requirements, we select a small XGBoost [6] model. Small XGBoost models are fast to train, and [6] provides an optimized C/C++ inference library that meets Auto-WLM's strict latency requirements.

Modelling procedure. Similar to past work [33], the first step in our modelling process is transforming a physical query plan tree into a feature vector. While this may not provide the optimal inductive bias for a performance model [28], we found this simple approach to be effective. Our features are not substantially different from those used in [33]: we walk the query plan tree and collect operators of the same type, and aggregate (count and sum) their estimated cost and cardinality. We additionally add features for the presence of operators that we know will have a significant effect on query performance (e.g., broadcast, reshuffle, range restricted scans), as well as the query type (e.g., `SELECT`, `INSERT`). While simple, these features can be collected quickly with a linear time walk of the query plan tree, using only statistics that were already computed by the optimizer.

As queries are executed, their observed features and latency are added to our training set. In order to ensure the training set does not grow unbounded,[1] we use a sliding window approach, discarding the oldest data point when a new one arrives. After initially implementing this approach, we realized that our training set was dominated by short queries – this makes sense, because the majority of queries executed on Amazon Redshift complete in less than 2 seconds. This can result in catastrophic predictions for long running queries (which were always estimated to be short, as the

[1]Keeping the training set small is also essential for reducing training overhead.

entire training set contained only short queries). To compensate for this, we partitioned our training set into n query latency "bins", each growing in size (e.g., the first bin may contain queries with latency between 0 and 10 seconds, while the second bin would contain queries with latency between 10 and 30 seconds). This way, a flood of short-running queries will only evict queries from the first bin, while rare long-running queries are kept around for longer.

At fixed intervals, the ML trainer fits an XGBoost model to the training set. We selected hyperparameters for XGBoost via a large experiment on the entire fleet, but found that the quality of the model was surprisingly insensitive to most hyperparameters. As a result, we selected hyperparameters primarily to reduce inference time (smaller models are faster).

Why not use the optimizer cost model? A natural question to ask about this architecture is: "you already have a query optimizer that is computing advanced statistics about the data in an optimized way, why not simply use the optimizer's cost estimate to predict query resources?" Indeed, this approach has been suggested before [53]. In fact, it seems quite odd that a model which takes optimizer cost as input can produce results better than the optimizer cost itself. There are several reasons why optimizer costs could be unfit for this purpose: (1) optimizer costs are unitless, and only comparable to themselves, (2) optimizer costs generally boil a complex query plan down to as single number using simple polynomials, (3) the optimizer's cost model is often the first thing "hacked" to encourage the planner to choose nodes taking advantage of new execution engine features, and (4) the optimizer's cost model is not instance-optimized to the user's data (e.g., the performance model can learn to correct the optimizer's mistakes, which will be different for each customer).

In the end, we found that a model trained on optimizer cost simply did not perform as well as the XGBoost model built from the query plan tree. This result is especially intuitive when one considers memory prediction: the optimizer's cost model will report similar costs for two query plans expected to have similar latency but drastically different memory usage. This is consistent with prior work [31, 33] as well. We validate this via an explicit comparison in Section 5.4.

What about when the optimizer statistics are bad? A serious concern with a model built on top of optimizer statistics is the accuracy of the optimizer's statistics. We know that, especially for queries with many joins, an optimizer's statistics may essentially be noise [24]. Luckily, since the query optimizer is deterministic, the noise produced for a particular join query will be *the same noise each time the query is seen.* This allows the XGBoost model to learn an association between a particular noisy value and a query runtime. Since a low-impact tweak to a query's predicates will (hopefully) only produce a low-impact change to the cardinality estimation, this learning is not just memorization (although memorization is part of it).

Unfortunately, there are circumstances when the optimizer's estimates are not just poor, but entirely non-existent. For example, a `COPY` query or a `SPECTRUM` query (arbitrary S3 read) depend on external data sources that Amazon Redshift does not (and in many cases, cannot) collect statistics about (e.g., the selectivity of a predicate). In these cases, in the absence of any good data, the ML predictor simply returns an estimate of the 95th percentile of the past queries of these types (e.g., for a `COPY` query, return the 95th percentile latency of all `COPY` queries seen previously). This strategy is not as bad as it might initially seem: by overestimating the time and resource requirements of these queries, the ML predictor essentially ensures they have adequate resources for execution.

A mutli-purpose model: The ML predictor is a critical component of Auto-WLM, but the ML predictor's design makes it usable throughout the rest of the database as well. Since the ML predictor can estimate the memory and time requirements for an arbitrary query, the ML predictor can be used for tasks such as materialized view refresh latency prediction, estimating the appropriate compiler optimization level for a query, and even estimating the impact of potential physical design changes. While a more purpose-built model might have been sufficient (and perhaps even superior) for Auto-WLM's purpose, this design allows the ML predictor to be a "service" for the rest of the cluster as well.

4.2 Priorities & admission controller

Estimating the performance properties of a query is only half the battle. Once estimates are computed, Auto-WLM uses a combination of a query prioritizer and an admission control module to decide when and where to execute queries.

After the query resource requirements (memory and CPU) and latency are estimated, the query is passed to the Priority Assigner. The Priority Assigner uses the estimates from the ML predictor to prioritize the query as either (1) short and cheap, (2) short, or (3-5) one of three coarse-grained levels of non-short queries (e.g., medium, long, extra long). After this prioritization is performed, the query is passed to the Admission Controller.

The Admission Controller is the most complex component of Auto-WLM, so we give a short overview of it here before proceeding to explain the details in the following subsections. Once the query is passed to the Admission Controller, Auto-WLM will first determine if the query is short running. If the query is predicted to be short running, the query is eligible for *Short Query Acceleration* (SQA), which allows the query to be executed on a dedicated slice of resources used only for short queries. Details about SQA are in Section 4.2.1. If a query is not short, Auto-WLM next attempts to find an open "slot" on either the user's main cluster or a concurrency scaling cluster that can execute the query, prioritizing the main cluster followed by the oldest concurrency scaling cluster. The number of slots available on each cluster, and the algorithm for choosing amongst them, is described in Section 4.2.2. If no slot is available, the query is entered into a priority queue (and may trigger horizontal scaling), which is described in Section 4.2.3.

4.2.1 Short query acceleration (SQA). Most Amazon Redshift clusters have a substantially skewed distribution of query latency: most queries are short, but a few queries are very long. As a result, if we applied a simple round-robin scheduling algorithm, every slot on a cluster could potentially become blocked by a long-running query. As a result, users would experience "head-of-line blocking" – a large number of short queries might be stuck waiting on a small number of long queries. This degrades average query latency.

To resolve this, Auto-WLM implements Short Query Acceleration (SQA). SQA reserves a small amount of resources on the user's main cluster that is always dedicated to executing short queries. Queries that are predicted to require only a small amount of resources and are expected to execute relatively quickly are eligible to be executed using these dedicated resources. If the performance model is incorrect, and a query executing using SQA resources ends up using significantly more time or resources than expected, the query is cancelled and reprocessed as a long query. This cancellation is transparent to the user.

There are many edge cases and failure modes for this design. For example, if the predictor produces a large number of mispredictions at once, the small dedicated SQA resources may become overwhelmed, leading to thrashing. Such thrashing would result in truly-short queries being evicted from SQA, resulting in massive requeuing. One might think that a solution to this problem would be to simply let mispredicted queries execute until they are complete. However, since queries executed using SQA-reserved resources are allocated less memory than normal, mispredicted queries can end up spilling to disk, potentially thrashing the local disk's cache and slowing down the entire cluster. Instead, Auto-WLM simply limits the maximum number of queries allowed to use SQA resources at once. This ensures that such cascading failures rarely occur.

Another important detail is the exact thresholds used for determining if a query is "short" or not. For example, imagine Alice is an Amazon Redshift customer for whom a five second query is considered short. But, for Bob, another Amazon Redshift customer, a one second query is short. If the short query threshold is set to 5 seconds, Alice is happy, but all of Bob's one second queries might execute with 5 seconds of latency due to head-of-line blocking. On the other hand, if the short query threshold is set to one second, Bob is happy, but Alice's short queries do not benefit from SQA at all. Auto-WLM solves this problem with a simple heuristic: for each cluster, we determine the 70th percentile execution time seen each week and use that as the short query threshold. Of course, this heuristic does not work for every customer, but it works for the majority of the Amazon Redshift fleet.

For users that have relatively long short queries (e.g., for customers where a short query can be defined as around 20 seconds), Auto-WLM makes an additional optimization called Super Short Query Optimization. The resources dedicated to SQA are split into a small piece and a large piece. The small piece is used to execute queries expected to take less than 5 seconds, and the large piece is used to execute queries expected to take between 5 and 20 seconds. This ensures that even for users with few short queries, very short queries still execute with increased priority.

The final optimization built into SQA is useful in the scenario where a cluster sees an exceptionally large number (say, hundreds) of short queries waiting for long queries to finish. In this case, the large number of short queries will be waiting for the SQA resources while the cluster is busy executing long queries. When this scenario is observed, Auto-WLM temporarily doubles the amount of SQA resources available by borrowing resources from long-running queries until the backlog of short queries is exhausted. This optimization increases queue time of long queries marginally while reducing the queue time for short queries by a large extent. Sometimes, this optimization leads to increase in execution time of both long and short queries due to increased contention for CPU resources, or if there are not enough short queries to justify the doubling. Therefore, this optimization is restricted to be active no more than 25% of the time.

4.2.2 Query placement & multiprogramming level. When a query is ineligible for Short Query Acceleration, the query is added to the queue, to be executed on either the main cluster or a concurrency scaling cluster. The exact organization of this queue is described in Section 4.2.3. Here, we will describe how Auto-WLM decides (1) the cluster onto which to place an incoming query, (2) the multiprogramming level of each cluster, and (3) when to spin up or spin down concurrency scaling clusters.

Placement. Auto-WLM uses a simple algorithm to place an incoming query onto a node: if there is an available slot with sufficient resources on the main cluster, place it there. Otherwise, check for an open slot with sufficient resources starting with the oldest concurrency scaling cluster. This approach imitates a greedy first-fit algorithm for the online bin packing problem. The main cluster and the oldest concurrency scaling clusters are checked first so that as the query workload gets smaller, other concurrency scaling clusters can be released.

Multiprogramming levels. Each cluster has a certain number of slots available for concurrent query execution. While many parallel DBMSes use a fixed multiprogramming level (number of concurrent query executions), Auto-WLM uses an adaptive algorithm to increase or decrease the multiprogramming level in response to workload changes. Auto-WLM's algorithm is similar to prior work [45], and depends on Little's Law, a fundamental theorem of queuing theory. In terms of query execution, Little's Law states:

$$C = T\hat{E} \tag{1}$$

where C is the number of executing queries in a stationary system (i.e., the distribution of the arrival rate of queries and the execution time of queries is not changing), T is the throughput of the system (queries per minute) and $\hat{E}$ is the average query execution time.

Using Equation 1, Auto-WLM can perform a what-if analysis to find out if admitting another query will increase throughput. To compute this, we assume that adding an additional query will linearly slow down existing queries. Thus, if we admit one more query with execution time E, we have a new number of executing queries $C' = C + 1$ and a new average execution time:

$$E' = \frac{\frac{(C+1)}{C}E + C\left(\hat{E} + \frac{1}{C}E\right)}{C+1}$$

Of course, the assumptions behind the above what-if analysis are valid only when cluster's CPU resources are a limiting constraint (e.g. we are in the linear slow down regime). While CPU resources are normally the limiting constraint, if a cluster is constrained on another factor, such as memory, the slow down could be super-linear. Similarly, if the cluster is not constrained by any resource, then there could be no slow down at all from adding an additional query. Note that when resources are available, Auto-WLM will admit the query.

From E', we can compute the throughput rate T' we would have if we admitted the new query as:

$$T' = \frac{C+1}{E'} \tag{2}$$

If T' is higher than T, we admit the new query by adding another concurrent slot. Note that by Equation 2, we are more likely to increase concurrency if the incoming query is short (because of the factor of E in the denominator). If we instead consider decreasing the multiprogramming level, note that we can compute T' in the same way, except we can use actual observed latency over a time window instead of predictor estimates.

Since we compute the hypothetical new throughput rate for an increase in concurrency using estimated values, but compute the hypothetical new throughput rate for a decrease in concurrency using observed values, it is possible that *both* an increase and a decrease in concurrency would seem to increase throughput. To resolve this issue, we always prioritize decreasing the concurrency level.

With this system, Auto-WLM is able to handle spikes of queries immediately without causing long queuing delays. Unfortunately, it is still possible that some medium-length queries with heavy resource requirements can cause the system to live lock. This occurs when there is a sudden shift from medium-running lightweight queries to medium-running heavy-weight queries. Auto-WLM checks this state by checking if any of the following conditions are met: (1) current memory and CPU usage is high across all clusters, (2) system concurrency is high across all clusters, and (3) all currently executing queries appear to be medium-length. When all of these conditions are true, Auto-WLM enters *emergency mode*. In emergency mode, Auto-WLM switches to a purely shortest-job-first scheduling policy using fixed concurrency levels. Once any of the three conditions are false, Auto-WLM switches back into its regular mode.

Elasticity. Auto-WLM is capable of handling many types of workload spikes via parallelism, but sometimes a query workload will still exceed the cluster's capacity. When a cluster begins to experience significant queuing (queries waiting for more than α seconds), Auto-WLM can spin up new concurrency scaling clusters and distribute queued queries to those clusters. This offers customers a cost/performance trade-off: customers can (a) accept queuing and thus lower performance, but see no cost increase, or (b) customers can allow Auto-WLM to scale out, maintaining consistent performance which may increase costs. Once a concurrency scaling cluster has been idle for a while (more than β seconds), the extra cluster is detached and the customer is no longer billed. We currently use fixed values for $alpha$ (60 seconds) and β (four minutes), which have been determined through a set of experiments across the fleet.

4.2.3 Queuing. With an adaptive multiprogramming level and concurrency scaling, significant queuing should be rare. However, customers often restrict the maximum amount of concurrent resources to control cost. Whenever concurrency is restricted additional queries may need to wait in the queue.

Auto-WLM's solution to maintaining query performance in the presence of heavy queuing is query prioritization. Auto-WLM supports six levels of user-specified priority: critical, highest, high, normal, low, lowest. Auto-WLM picks queries to execute using a weighted round-robin algorithm [5, 19]. Intuitively, the weighted round-robin algorithm assigns each query a value proportional to the query's priority, and then selects a query by sampling from the resulting probability distribution. Since every query has a non-zero probability of being selected (even those with low priority), Auto-WLM gets starvation control "for free."

Preemption. The weighted round-robin algorithm works well when queries with mixed priority are queued. However, if the cluster is occupied by long-running low-priority queries and a high-priority query arrives, the high-priority query will wait on the low-priority queries. To remedy this, Auto-WLM supports preemption of lower priority queries: when a low-priority query is preempted, the query is cancelled and re-queued to be executed later, freeing up cluster resources for higher priority queries. Because low-priority queries are cancelled, system resources are wasted — imagine if a long-running low-priority query was cancelled seconds before it completed. The entire computation must now be redone. Auto-WLM deals with this issue in two ways.

First, a cool down period: when a low-priority query needs to be preempted, Auto-WLM chooses the query with the longest remaining execution time (based on the predicted execution time of the query). If a preempted query begins to re-execute and is once again selected for preemption, Auto-WLM enters a short "cool down" period before preempting the query for a second time. Each time a query is preempted, this cool down period increases exponentially, ensuring that the query eventually has time to finish. Second, *long term guardrails*: although the cool down period ensures that *any* query will eventually get to complete, the total amount of waste (time spent processing a query before it was preempted) is still unbound across *all* queries. To ensure that the majority of a cluster's resources are not wasted, Auto-WLM employs *waste-time bounded preemption* (WBP). Within a time window, WBP tracks the total amount of wasted time W from preemption and the total amount of useful work U from queries that completed. If the ratio of wasted work to useful work $\frac{W}{U}$ exceeds a threshold, Auto-WLM stops preempting queries entirely.

By combining weighted-round robin scheduling with smart preemption, Auto-WLM is able to provide intelligent query scheduling while simultaneously ensuring liveness and a limit on the amount of resources wasted via preemption. While Auto-WLM may not be the *optimal* solution, we found the strategy of combining model predictions with traditional heuristics to be effective for Amazon Redshift. We believe that Auto-WLM strikes a great balance between pragmatism and innovation.

5 EXPERIMENTS

In this section, we experimentally evaluate the quality of Auto-WLM's decisions, along with the accuracy of Auto-WLM's internal query predictor. Specifically, we evaluate:

(1) **Concurrency scaling (Section 5.1)**: How well can Auto-WLM control the number of concurrent query executions and maintain good query latency and throughput?
(2) **SQA, short query acceleration (Section 5.2)**: How well can Auto-WLM differentiate between short cheap queries

and long expensive queries, and what impact does SQA have on query latency?

(3) **Bursting (Section 5.3)**: How well can Auto-WLM provide linear scaling via concurrency scaling clusters?

(4) **Predictor accuracy (Section 5.4**: How accurate is Auto-WLM's query resource model, the core component used to make intelligent decisions?

Unless stated otherwise, all experiments are conducted using a four node Amazon Redshift cluster. Each node has 32 vCPU cores and 244 GB of RAM (`dc2.8xlarge`). The cluster is configured to use all four nodes as the base cluster. For workloads, we use a combination of TPC-DS at three different scale factors (1G, 100G, and 3T) and data from the Redshift fleet. TPC-DS queries were generated and executed using the open source Cloud Data Warehouse Benchmark[2]. All TPC-DS queries are run with 30 streams initialized with different seeds. Redshift fleet data is taken from a sample of 10,000 clusters alive for at least three months as of October 1st, 2022.

5.1 Concurrency adjustment

One of the hardest parts of configuring an analytics database is deciding how many queries to execute at once (the multiprogramming level). If the multiprogramming level is set too low, cluster resources are left underutilized, negatively impacting throughput. If the multiprogramming level is set too high, queries will contend with each other for scarce resources, negatively impacting latency.

Auto-WLM automatically selects a multiprogramming level for the user's cluster adaptively, as described in Section 4.2. To evaluate the quality of Auto-WLM's concurrency scaling decisions, we evaluated three scales of TPC-DS at 50 different multiprogramming levels (1 to 50). Here, we discuss results for multiprograming levels 5, 10, and 20, which are the three we found to be optimal at the 99th percentile for the tested TPC-DS scale factors.

The results are plotted in Figure 3. We plot average query latency (`Avg`), along with the 50th, 90th, and 99th percentile of latency. Note that a lower multiprogramming level is not always optimal for overall query latency because of queuing: in the extreme case, if only one query is executed at a time, the other queries will incur large queuing delays. For each scale factor, Auto-WLM is able to achieve optimal or near-optimal latency at each percentile. The largest difference between Auto-WLM and a manual configuration is in P99 latency for the largest workload, where Auto-WLM performs significantly worse than MP20. This is because Auto-WLM chooses to trade some tail latency for significantly improved median latency. *Thus, Auto-WLM can automatically set an appropriate multiprocessing level for workloads at multiple scales, relieving the user from a complex and resource-intensive tuning process.*

5.2 Short query acceleration (SQA)

Auto-WLM offers short query acceleration (SQA), a service that automatically sends queries with low resource requirements and low expected latency to a special, fast queue (the short query queue). When a query is added to the short query queue, it will be executed with a limited amount of resources. If a query in the short queue does not finish in a short amount of time, the query is "timed out" and placed in the main queue to be executed later.

[2]https://github.com/awslabs/amazon-redshift-utils/

	TPC-DS 100G		TPC-DS 3T	
	Count	Percent	Count	Percent
Exec in SQA	4256	82%	898	16%
Timed out in SQA	126	2%	267	5%
Exec in long queue	938	18%	4772	84%

Table 1: SQA results for 100G and 3T scale TPC-DS. Incoming queries are classified as either short or long. Long queries are sent directly to the long queue. Short queries are first placed in the SQA queue. If a query in the short queue times out, it is moved to the long query queue. Percentages add up to over 100% because queries that are mistakenly placed in the short queue time out and are then fully executed in the long queue. SQA has a low false-positive rate, erroneously classifying only 2% and 5% of queries as short that should have been classified as long.

Impact on query latency. By letting short queries access an exclusive resource reservation, shorter queries can be executed first, improving the latency of short-running queries. Figure 4 shows the impact SQA has on the distribution of query latency for TPC-DS 100G. The 40th percentile improves by over 90%, while the maximum (P100) latency regresses by only 20%. Of course, whether or not this trade-off is worthwhile depends on the user's workload. While we have not conducted a formal user survey, we know from limited feedback that many users are happy with this option.

Short query classification accuracy. SQA relies on the query performance model to predict whether or not a query will be short running, and thus eligible for the short queue, or whether it will be long running. If this model is accurate, Redshift will emulate a shortest-job-first schedule, decreasing average latency. However, if too many queries are erroneously placed in the short queue, resources will be wasted executing, timing out, and re-executing mispredicted queries in the long queue. Thus, it is critical that only queries that are truly low-resource and low-latency are placed in the short query queue. To measure this, we executed TPC-DS 100G and 3T[3] and recorded the number of queries placed into the short queue that fully executed and the number that timed out. The results are displayed in Table 1. For TPC-DS 100G, Auto-WLM only misclassified 2% of queries as short-running that were truly long-running, which means that only 126 queries needed to be re-executed in the long queue after being timed out. For TPC-DS 3T, the false positive rate of the short query classifier was 5%, resulting in 267 re-executions.

Overall, we conclude that Auto-WLM's classifier is sufficiently accurate to differentiate between long and short running queries in a way that reduces latency for short running queries. We more fully evaluate the accuracy of Auto-WLM's predictor in Section 5.4. *Auto-WLM's SQA functionality provides users with an easy and automatic way to reduce query latency for workloads by prioritizing short-running queries.*

[3]We do not report results on TPC-DS 1G, because nearly every query could be executed in the short queue.

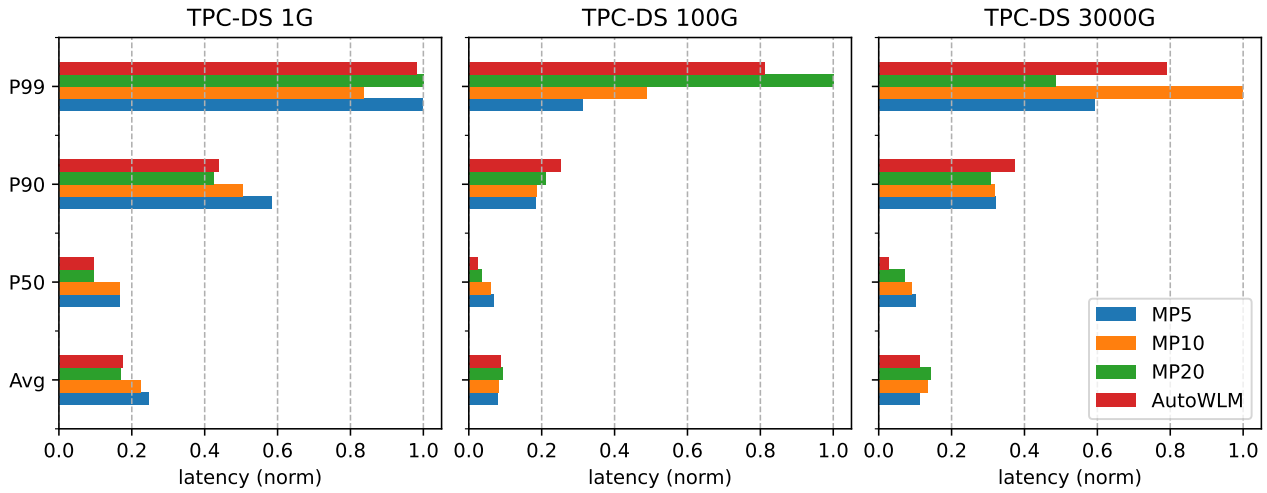

Figure 3: Auto-WLM concurrency scaling compared with 4 different manual concurrency levels for three different TPC-DS scales. MP5, MP10, and MP20 represent fixing the multiprogramming level to a specific value. Auto-WLM dynamically selects the multiprogramming level. Latency is normalized to the largest value in each plot. Average latency, median latency (P50), the 90th percentile (P90), and the 99th percentile (P99) of query latency are shown. While the optimal multiprogramming level is difficult to determine, Auto-WLM is able to automatically choose a multiprogramming level that is optimal or near-optimal for each scale factor.

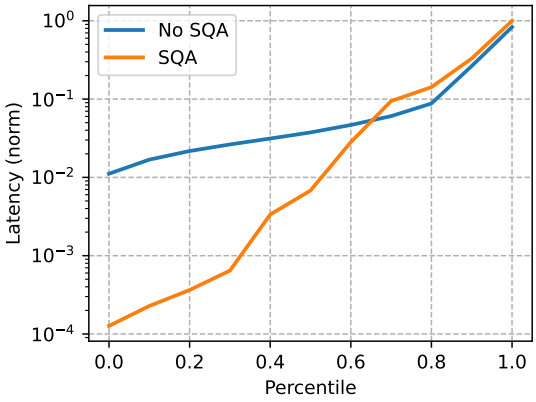

Figure 4: Distribution of query latency for TPC-DS 100G with and without SQA. When SQA is enabled, queries predicted to be short-running are placed into a special queue. This results in a decrease in latency for short running queries, but an increase in tail latency.

Figure 5: Behavior of Auto-WLM concurrency scaling compared to a manual baseline for TPC-DS 100G. The x-axis represents the maximum number of concurrent clusters Auto-WLM is allowed to create (with zero meaning that horizontal concurrency scaling is disabled). The left plot shows the throughput of the cluster, which increases as the number of allowed concurrency scaling clusters increases. The right plot shows the total number of queries assigned to concurrency scaling clusters at each configuration.

5.3 Elasticity

Next, we evaluate the elasticity component of Auto-WLM. When configured to do so, Auto-WLM will spin up new concurrency scaling clusters to accommodate a spike in the query workload. Each cluster's multiprogramming level is managed by Auto-WLM, and queries are distributed based on each cluster's available resources.

Figure 5 compares the throughput of Auto-WLM with a *manual* baseline, which uses uses a fixed number of concurrent clusters (0 to 10) at any point in time and a fixed multiprogramming level of 10. For *manual*, each incoming query is executed on the first cluster with an available slot (if no slots are available, the query waits in the queue). The main cluster is considered the "first" cluster.

When a small number of concurrency scaling clusters are available (less than 6), Auto-WLM efficiently manages the multiprogramming level on each cluster to maximize throughput, achieving significantly higher throughput than the manual baseline. Once a large number of concurrency scaling clusters are available (more than 8), the performance of both approaches are comparable: once enough concurrency scaling clusters are available, both approaches essentially spread out queries evenly. Thus, *Auto-WLM can achieve significantly more throughput with fewer clusters (and thus lower cost) than our manual baseline.*

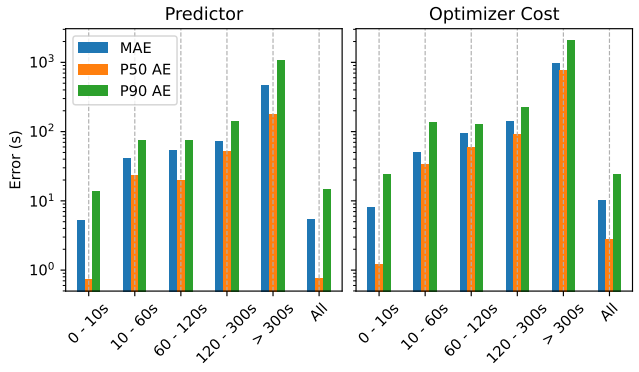

Figure 6: Absolute error comparisons between the Auto-WLM query predictor (left) and simple linear regression with optimizer cost (right). Mean absolute error (MAE), median absolute error (P50), and 90th percentile absolute error (P90) are plotted. Note the log scale. Auto-WLM predictions are significantly better than linear regression on optimizer cost, especially in the tail.

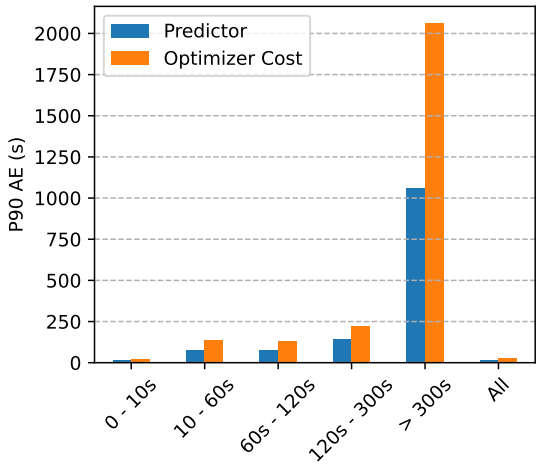

Figure 7: A direct comparison of 90th percentile absolute error between the Auto-WLM predictor and linear regression with a cost model. Bars here are at absolute scale compared to those on a log scale in Figure 6.

5.4 Predictor accuracy

In this section, we evaluate the performance of Auto-WLM's internal query performance model, or query predictor. Almost all of Auto-WLM's features are dependent on accurate query performance modeling, which is known to be a difficult task. In addition, Auto-WLM's trains its predictor on the user databases, requiring that both the training and inference procedure be cheap to execute; typically, Auto-WLM models provide sub-millisecond inference and train in only a few dozen milliseconds. As a result, comparing the performance of Auto-WLM's model accuracy to heavyweight techniques like QPPNet [31] or TCNNs [34] is not appropriate. Instead, we compare the accuracy of Auto-WLM's predictor with a simple linear regression against the optimzer's cost model.

Here, we depart from using TPC-DS data. Since TPC-DS only contains 99 query templates, learning their execution time is trivial. Instead, we evaluate Auto-WLM's predictor on a set of 5 million queries sampled from Amazon Redshift clusters. The results are plotted in Figure 6. We split our analysis between five different query latency "bins". This is critical because the vast majority of queries are short: if one were to train a naive model across all the data, one could achieve seemingly-reasonable accuracy by simply guessing the median. Across each bin, we report the mean absolute error (MAE) along with the median (P50) and 90th percentile (P90) of absolute prediction error. For queries with latency between one and two minutes, the Auto-WLM predictor has a median error of 20 seconds, compared to the optimizer cost regression with a median error of 55 seconds.

We plot the 90th percentile errors for each bucket without a log scale in Figure 7. Here, the relationship between query runtime and error is clear: longer queries are "harder" to predict (i.e., exhibit higher prediction error). For queries taking longer than five minutes, Auto-WLM's predictor has a 90th percentile error of around 1000 seconds, which for many queries is longer than their runtime. Other, more advanced query performance prediction models could provide better accuracy, especially at the tail, but often require significant larger inference time (see also Section 6)

Taken as a whole, Auto-WLM's predictor has reasonably low error across the Amazon Redshift fleet (e.g., the "All" column in Figure 6). As we have shown in previous sections, these predictions are of suitable accuracy for many applications and improvements to the predictor's accuracy should only make such downstream tasks more efficient. *Thus, while Auto-WLM's predictor might not have perfect accuracy, the predictor is able to sufficiently differentiate between queries for our applications.*

6 LESSONS LEARNED AND HOW THE RESEARCH COMMUNITY CAN HELP

While there have been hundreds of ML for systems papers over the last few year, to the best of our knowledge, Auto-WLM is one of the first ML-enhanced component deployed in a large-scale customer-facing data-warehouse services, which goes beyond logical-tuning (e.g., index or materialized view recommendation). While building Auto-WLM we not only learned several valuable lessons on how to make ML-enhanced components actual work, but also found a significant gap in what academic papers address and what is required in practice. In the following, we describe several of these lessons and outline research directions we believe are currently under-served.

6.1 Global vs. local training and transferable models

Currently, Auto-WLM uses exclusively locally-trained models: each cluster trains its own performance model using the cluster's workload as a training set (see Section 4.1). Unfortunately, this creates a cold-start problem for new customers or clusters; we do not have a model before sufficient local training data exists. The obvious solution is that we should build transferable and/or zero-shot models [16] by training the models globally across clusters. Not only would one expect that those models would be far superior as more

training data is used, but that they would also solve the cold-start problem. Yet, to our surprise, the globally trained models performed significantly worse than our locally trained models.

We believe that this can be explained by the fact that the global model can at best learn a good (transferable) cost model but is not able to learn anything about the data of the individual clusters – in other words, the global model is unable to "instance optimize" to any one database. Intuitively, this is because learning a query latency predictor for *any* specific database with a fixed workload is easier than learning a query latency predictor for *every* database. While a more precise cost model can help with better predictions, it is entirely over-shadowed by what the local models are able to learn about the data and workload to make better estimates. This is also in line with the recent work [38, 39].

Finally, we observed some promising results on training deep learning models locally. A locally trained deep learning model was able to outperform decision tree models, especially on long-running queries that had not been previously seen. However, we note that (1) deep learning methods had far too much overhead to be used in Auto-WLM, and (2) query workloads tend to be highly repetitive: over the entire Amazon Redshift fleet we observed that up to 75% of all queries have been seen before. Boosted decision trees are great at memorizing previously seen queries and provide just enough generalization for the unknown queries to make them work surprisingly well.

Where does this leave us? Are the more advanced deep learning-based methods all academic and not practical? Based on our observations so far, we believe the impact of *transferable data-independent* cost models will be limited. At the same time, training *data- and workload-dependent* query time prediction models or *cardinality-correction* models (as proposed in [39]) show the most promise but might be too expensive to train for each individual cluster. Unfortunately, *data-dependent but workload-independent* models (e.g., learning data statistics independent of the workload) appear to be the worst of both worlds as they waste a lot of model capacity on data properties, which might not be relevant - recall that workloads at least on Amazon Redshift tend to be highly repetitive.

Hence, we believe the most impactful area of research is in techniques which allow to quickly adapt and efficiently learn the most important data characteristics and performance characteristics for the most common query patterns, in a low-overhead way. While queries are not necessarily exactly the same, it is very rare that a query contains, for example, a never-observed join path. Similarly, training global models, which are then locally refined might be an option, though initial experiment show that the "refinement" in some cases almost takes as long as training from scratch.

6.2 Inference time, model size, and one-size doesn't fit all

In theory, a single model should be able to predict the query latency and resource requirements for each query. However, in practice it is not only hard to build a single model which does it all, but one large model is also unlikely to meet the inference and training performance requirements. We observed that the majority of all queries are short running and take under 10s. For short running queries, a complex neural-net based model would add CPU and latency overhead. Particularly for dashboard queries, which often run under 1s, this poses a significant challenge.

Thus, we suggest that the academic research community should not only focus on building the most accurate models, but also closely consider resource overhead and latency. It is simply not enough to show that a model improves q-error or average query latency, which is often dominated by the long-running queries. We must also carefully consider that each query-class (e.g., short-running dashboard queries) are not negatively impacted. One of the easiest ways to do this is to test the *end-to-end* impact of a predictor on a system – that is, instead of simply measuring prediction accuracy, integrate the predictor into a database component and measure the increase (or decrease) in actual performance. One potential solution to the problem is to combine different type of models and to train (simpler) models for specific tasks.

6.3 The long-tail, fall-backs, and guardrails

Many recent papers about ML-enhanced components [20, 28, 54] demonstrate how overall performance improved at the cost of very few regressions in the long tail. Unfortunately, at the scale of Amazon Redshift even a few regressions might significantly impact a large group of customers. Generally speaking, this is not a reason that a technique should not be deployed, as it is rarely the case that one technique is strictly superior to another (e.g., if a new technique improves most queries but regresses a few, then all the improved queries may be considered "pre-regressions" in the current system). However, it is important to determine upfront how the regressions are handled; a question often ignored by the academic community.

For example, is it possible to detect the regressions and gracefully fall-back to the pre-existing behavior? This leads to a different set of research questions such as: what guardrails can be added to avoid the worst-case behavior? What statistics and tools can we give the support team to debug the model and algorithm decision? How can the operational team override the policy if needed? Unfortunately, it is not always easy to add such protection methods – but without at least some protection, it is essentially impossible to deploy any ML-enhanced component.

6.4 Explainability

Related to the previous argument, making the decisions of an automated or ML enhanced system explainable is in many cases critically important. For example, concurrency scaling decisions have a direct impact on the cost for customers and customers want to know that they are charged correctly. As long as everything works as expected, most customers do not care about the individual decisions. However, when something goes wrong (e.g., cost increases, or a concurrency scaling cluster is not created), we need insights into why a decision was made and a direct way to fix it.

6.5 TPC-X workloads are not representative

While not a big secret, most existing benchmarks are far from being representative of what we observe in real workloads. Queries are more diverse across clusters, more repetitive, have workload peaks and troughs, consist of mixed workloads (e.g., ETL, dashboarding, and exploration queries) and contain a significant number of writes,

among other features missing in benchmarks. While Amazon Redshift uses TPC-X workloads as sanity checks, it is unclear if TPC-X is even representative for a single customer workload beyond proof-of-concept evaluations. While there has been some recent attempts to create more realistic benchmarks [3], much more work is needed.

Addressing this problem will require creative solutions between industry and academia. For example, we need new techniques which help to replicate anonymized workloads (not data), which customers and cloud providers accept. Are there ways to create and share benchmarks that does not require one to share data, queries, or schema? This requires research into methods to create a synthetic workload and data set, which mimics the performance characteristics of the original database. This should be much easier than differential privacy, which many in the industry often are not comfortable with.

6.6 Simplicity over Complexity

While reinforcement learning is extremely powerful and has shown to yield impressive results [25, 44], it is also very complex and hard to get right. Thus, at Amazon Redshift, we always start with the simplest models and heuristics and iterate quickly before trying the state-of-the-art, often highly complex, techniques. For example, our approach so far has shown that even simple decision trees are able to achieve sufficient performance for many applications, all while being significantly simpler and lower overhead than tree convolution networks [28, 34]. Hence, we urge the research community, especially future reviewers, to value simplicity over unnecessarily complex solutions, which tries to squeeze out the last bit of accuracy often at the cost of never being really usable.

6.7 Synergy effects

In SageDB [22], we proposed that the same type of models will be shared across components. While originally just a vision, this is already becoming reality at Amazon Redshift. Models are shared across Auto-WLM and other ML-enhanced components (e.g., Automatic Materialized View creation). Obviously, this has enormous advantages in regard to training cost and development time.

These synergies also opens up new opportunities and challenges. For example, not every component uses the model's output in the same way. This rarely matters if the model would be perfect, but does matter with errors as they are often not uniformly distributed. One component might only case if a query is short or long running, another cares only about the accuracy of repetitive long-running queries, another only about very large data lake queries, and so on. So far, the research community focused mainly on improving separate ML-enhanced components, leaving out many of the interesting research challenges which exist when models are more broadly used within a system.

7 CONCLUSION

Auto-WLM uses machine learning models to provide automatic scheduling, horizontal scaling, and admission control for Redshift users. Auto-WLM is deployed in production and schedules every single query on Amazon Redshift. Future work will focus not just on more advanced ML techniques, but also on systematic integration of models across the Amazon Redshift platform.

REFERENCES

[1] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, Sept. 1992.

[2] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig, E. Hotinger, Y. Leshinksy, J. Liang, M. McCreedy, F. Nagel, I. Pandis, P. Parchas, R. Pathak, O. Polychroniou, F. Rahman, G. Saxena, G. Soundararajan, S. Subramanian, and D. Terry. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 2205–2217, New York, NY, USA, June 2022. Association for Computing Machinery.

[3] L. Bindschaedler, A. Kipf, T. Kraska, R. Marcus, and U. F. Minhas. Towards a Benchmark for Learned Systems. In *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, SMDB @ ICDE '21, pages 127–133, Apr. 2021. ISSN: 2473-3490.

[4] R. Bordawekar and O. Shmueli. Using Word Embedding to Enable Semantic Queries in Relational Databases. In *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning (DEEM)*, DEEM '17, pages 5:1–5:4, 2017.

[5] H. M. Chaskar and U. Madhow. Fair scheduling with tunable latency: A round-robin approach. *IEEE/ACM Transactions on Networking*, 11(4):592–601, Aug. 2003.

[6] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[7] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[8] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri. Automatically indexing millions of databases in microsoft azure SQL database. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 666–679. ACM, 2019.

[9] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *38th ACM Special Interest Group in Data Management*, SIGMOD '19, 2019.

[10] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska. Tsunami: a learned multi-dimensional index for correlated data and skewed workloads. *Proceedings of the VLDB Endowment*, 14(2):74–86, Oct. 2020.

[11] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. *PVLDB*, 2(1):1246–1257, 2009.

[12] J. Duggan, O. Papaemmanouil, U. Cetintemel, and E. Upfal. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT '14, pages 109–120, 2014.

[13] P. Ferragina and G. Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, Apr. 2020.

[14] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *2009 IEEE 25th International Conference on Data Engineering*, ICDE '09, pages 592–603, Mar. 2009.

[15] J. R. Haritsa, M. J. Canrey, and M. Livny. Value-based scheduling in real-time database systems. *The VLDB Journal*, 2(2):117–152, Apr. 1993.

[16] B. Hilprecht and C. Binnig. Zero-shot cost models for out-of-the-box learned cost prediction. *Proceedings of the VLDB Endowment*, 15(11):2361–2374, July 2022.

[17] Hussam Abu-Libdeh, Deniz Altinbuken, Alex Beutel, Ed Chi, Lyric Doshi, Tim Kraska, Xiaozhou Li, Andy Ly, and Christopher Olston. Learned Indexes for Google-scale Disk-based Database. In *Machine Learning for Systems Workshop at NeurIPS 2020*, MLForSystems @ NeurIPS '20, Vancouver, BC, Canada, 2020.

[18] T. Kaftan, M. Balazinska, A. Cheung, and J. Gehrke. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. *arXiv preprint*, Feb. 2018.

[19] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on Selected Areas in Communications*, 9(8):1265–1279, Oct. 1991.

[20] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[21] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM @ SIGMOD '20, pages 1–5, Portland, Oregon, June 2020. Association for Computing Machinery.

[22] T. Kraska, M. Alizadeh, A. Beutel, Ed Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A Learned Database System. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[23] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management*

of Data, SIGMOD '18, New York, NY, USA, 2018. ACM.

[24] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, 2015.

[25] H. Mao, P. Negi, A. Narayan, H. Wang, J. Yang, H. Wang, R. Marcus, r. addanki, M. Khani Shirkoohi, S. He, V. Nathan, F. Cangialosi, S. Venkatakrishnan, W.-H. Weng, S. Han, T. Kraska, and M. Alizadeh. Park: An Open Platform for Learning-Augmented Computer Systems. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d. Alche-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, NeurIPS '19, pages 2490–2502. Curran Associates, Inc., 2019.

[26] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. *arXiv:1810.01963 [cs, stat]*, 2018. arXiv: 1810.01963.

[27] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, China, June 2021. Award: 'best paper award'.

[28] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A Learned Query Optimizer. *PVLDB*, 12(11):1705–1718, 2019.

[29] R. Marcus and O. Papaemmanouil. WiSeDB: A Learning-based Workload Management Advisor for Cloud Databases. *PVLDB*, 9(10):780–791, 2016. tex.acmid= 2977804 tex.issue_date= June 2016 tex.numpages= 12.

[30] R. Marcus and O. Papaemmanouil. Releasing Cloud Databases from the Chains of Performance Prediction Models. In *8th Biennial Conference on Innovative Data Systems Research*, CIDR '17, San Jose, CA, 2017. tex.authors= Ryan Marcus and Olga Papaemmanouil.

[31] R. Marcus and O. Papaemmanouil. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *PVLDB*, 12(11):1733–1746, 2019.

[32] R. Marcus, O. Papaemmanouil, S. Semenova, and S. Garber. NashDB: An End-to-End Economic Method for Elastic Database Fragmentation, Replication, and Provisioning. In *Proceedings of the 37th ACM Special Interest Group in Data Management*, SIGMOD '18, Houston, TX, 2018.

[33] Mert Akdere and Ugur Cetintemel. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 390–401. IEEE, 2012.

[34] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI '16, pages 1287–1293, Phoenix, Arizona, 2016. AAAI Press.

[35] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep Learning for Entity Matching: A Design Space Exploration. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 19–34, New York, NY, USA, 2018. ACM.

[36] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning Multi-dimensional Indexing. In *ML for Systems at NeurIPS*, MLForSystems @ NeurIPS '19, Dec. 2019.

[37] P. Negi, M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. Friedman, and A. Jindal. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2557–2569, Virtual Event China, June 2021. ACM. Award: 'best paper honorable mention'.

[38] P. Negi, R. Marcus, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Cost-Guided Cardinality Estimation: Focus Where it Matters. In *Workshop on Self-Managing Databases*, SMDB @ ICDE '20, 2020.

[39] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. Robust Query Driven Cardinality Estimation under Chaning Workloads. *PVLDB*, 16(6):1520 – 1533, 2023.

[40] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: a data platform based on the scaling-up approach. *Proceedings of the VLDB Endowment*, 11(6):663–676, Feb. 2018.

[41] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-Driving Database Management Systems. In *8th Biennial Conference on Innovative Data Systems Research*, CIDR '17, 2017.

[42] A. Pavlo, E. P. C. Jones, and S. Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *PVLDB*, 5(2):86–96, 2011.

[43] I. Sabek, T. S. Ukyab, and T. Kraska. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 1228–1242, New York, NY, USA, June 2022. Association for Computing Machinery.

[44] M. Schaarschmidt, A. Kuhnle, B. Ellis, K. Fricke, F. Gessert, and E. Yoneki. LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations. *arXiv:1808.07903 [cs, stat]*, Aug. 2018.

[45] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to Determine a Good Multi-Programming Level for External Scheduling. In *22nd International Conference on Data Engineering*, ICDE '06, pages 60–60, Atlanta, GA, USA, 2006. IEEE.

[46] Y. Sheng, A. Tomasic, T. Zhang, and A. Pavlo. Scheduling OLTP transactions via learned abort prediction. In R. Bordawekar and O. Shmueli, editors, *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, pages 1:1–1:8. ACM, 2019.

[47] Shrainik Jain, Jiaqi Yan, Thiery Cruanes, and Bill Howe. Database-Agnostic Workload Management. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[48] S. Tozer, T. Brecht, and A. Aboulnaga. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, ICDE '10, pages 397–408, Mar. 2010.

[49] I. Trummer, S. Moseley, D. Maram, S. Jo, and J. Antonakakis. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *PVLDB*, 11(12):2074–2077, 2018.

[50] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, New York, NY, USA, 2017. ACM.

[51] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 363–378, 2016.

[52] B. Wagner, A. Kohn, and T. Neumann. Self-Tuning Query Scheduling for Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 1879–1891, New York, NY, USA, June 2021. Association for Computing Machinery.

[53] W. Wu, H. Hacigumus, Y. Chi, S. Zhu, J. Tatemura, and J. F. Naughton. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable? In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 1081–1092, Washington, DC, USA, 2013. IEEE Computer Society.

[54] Z. Yang, W.-L. Chiang, S. Luan, G. Mittal, M. Luo, and I. Stoica. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 931–944, New York, NY, USA, June 2022. Association for Computing Machinery.

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

[56] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *2020 IEEE 36th International Conference on Data Engineering*, ICDE '20, pages 1297–1308, Apr. 2020. ISSN: 2375-026X.

[57] C. Zhang, R. Marcus, A. Kleiman, and O. Papaemmanouil. Buffer Pool Aware Query Scheduling via Deep Reinforcement Learning. In B. He, B. Reinwald, and Y. Wu, editors, *2nd International Workshop on Applied AI for Database Systems and Applications*, AIDB@VLDB '20, Tokyo, Japan, 2020.

[58] W. Zhang, M. Interlandi, P. Mineiro, S. Qiao, N. Ghazanfari, K. Lie, M. Friedman, R. Hosn, H. Patel, and A. Jindal. Deploying a Steered Query Optimizer in Production at Microsoft. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 2299–2311, Philadelphia PA USA, June 2022. ACM.