Making Data Clouds Smarter at Keebo: Automated Warehouse Optimization using Data Learning

Barzan Mozafari, Radu Alexandru Burcuta, Alan Cabrera, Andrei Constantin, Derek Francis, David Grömling, Alekh Jindal+, Maciej Konkolowicz, Valentin Marian Spac, Yongjoo Park, Russell Razo Carranza, Nicholas Richardson, Abhishek Roy, Aayushi Srivastava, Isha Tarte, Brian Westphal, Chi Zhang
Keebo, Inc.
Ann Arbor, MI, USA
research@keebo.ai

ABSTRACT

Data clouds in general, and cloud data warehouses (CDWs) in particular, have lowered the upfront expertise and infrastructure barriers, making it easy for a wider range of users to query large and diverse sources of data. This has made modern data pipelines more complex, harder to optimize, and therefore less resource efficient. As a result, the ongoing cost of data clouds can easily become prohibitively expensive. Further, since CDWs are general-purpose solutions that must serve a wide range of workloads, their out-of-box performance is sub-optimal for any single workload. Data teams therefore spend significant effort manually optimizing their queries and cloud infrastructure to curb costs while achieving reasonable performance. Aside from the opportunity cost of diverting data teams from business goals, manual optimization of millions of constantly changing queries is simply daunting.

To the best of our knowledge, Keebo's Warehouse Optimization is the first fully-automated solution capable of making real-time optimization decisions that minimize the CDWs' overall cost while meeting the users' performance goals. Keebo learns from how users and applications interact with their CDW and uses its trained models to automatically optimize the warehouse settings, adjusts its resources (e.g., compute, memory), scale it up or down, suspend or resume it, and also self-correct in real-time based on the impact of its own actions.

CCS CONCEPTS

• **Computing methodologies → Reinforcement learning**; • **Information systems → Database performance evaluation**.

KEYWORDS

Cloud Data Warehouse, Reinforcement Learning, Data Learning, Warehouse Optimization

ACM Reference Format:
Barzan Mozafari, Radu Alexandru Burcuta, Alan Cabrera, Andrei Constantin, Derek Francis, David Grömling, Alekh Jindal+, Maciej Konkolowicz, Valentin Marian Spac, Yongjoo Park, Russell Razo Carranza, Nicholas Richardson, Abhishek Roy, Aayushi Srivastava, Isha Tarte, Brian Westphal, Chi Zhang. 2023. Making Data Clouds Smarter at Keebo: Automated Warehouse Optimization using Data Learning. 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/10.1145/3555041.3589681

SIGMOD-Companion '23, June 18–23, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9507-6/23/06.
https://doi.org/10.1145/3555041.3589681

1 INTRODUCTION

Data clouds. Data clouds have radically changed how modern data applications are built, deployed and used. In particular, cloud data warehouse (CDW) offerings, such as Google's BigQuery [35], Amazon Redshift [23], Snowflake [22], and Azure Synapse [42], have significantly lowered the barrier in terms of the expertise and the upfront infrastructure investments required to develop data-intensive applications. In other words, a much wider range of users and applications to tap into data. Modern applications can now query larger data-sets combining a larger number of data sources, without worrying about the challenges of storing and processing volumes of data. With CDWs, compute and storage resources can be scaled up and down *on demand*, automating much of the manual work traditionally done by experienced data engineering and data ops teams (e.g., hardware provisioning, scaling, software updates, index and view creation). This evolution has fueled much of the rapid market growth. For example, the CDW market is expected to reach \$39B by 2026, with a 31% compound annual growth rate during 2021-2026. The overall market for big data and business analytics is also expected to reach \$152B by 2026 [40].

New challenges. Nonetheless, this lower barrier to entry—i.e., elimination of the upfront capital expenses (CapEx)—has had other consequences. The ease of sharing and querying various data sources, and simplified access to virtually unlimited compute resources, have led to new challenges. Most notably, modern data pipelines have become far more complex, harder to optimize, and therefore less resource efficient. For example, 20% of the companies in a recent survey use 1000 or more data sources [33], and data teams spend 56% of their time on DataOps according to another survey [3]. The additional complexity and the resulting computational inefficiencies have drastically increased the operational expenses (OpEx) for customers of data clouds in two ways. First, they now spend more on their cloud computing bill (i.e., infrastructure costs) [1, 41]. Second, their data teams now have to spend significant effort on

+Work done while at Keebo.

manually optimizing their data pipelines, applications, and queries in order to maintain acceptable performance while keeping their cloud bill within budget. These manual optimizations have a direct cost, as qualified data engineers are scarce and costly. There is also an indirect (opportunity) cost, as data teams are distracted from their mission of improving their own business through data to operating and tuning their data infrastructure. In summary, while data clouds have drastically reduced the upfront (CapEx) barrier, they have increased the ongoing (OpEx) barrier for their customers.

Need for constant and complex optimizations. The fundamental reason why CDWs still require significant tuning and manual optimizations by users is that CDWs are designed as general-purpose solutions that need to serve a wide range of use cases and workloads, such as ETL, adhoc analytics, reporting, Business Intelligence (BI), and even custom applications. As a result, their out-of-box experience is by definition sub-optimal for any single workload. Even the same type of workload differs vastly from one customer to another, and even between different warehouses of the same customer. In fact, new patterns continuously emerge as new applications or industries migrate to data clouds. Business needs and requirements constantly change over time too. For instance, a company in growth mode may prioritize performance and time-to-insight over infrastructure costs. However, the same company may focus on reducing costs when faced with a recession or economic downturn. To accommodate their wide range of users, each CDW exposes a certain number of knobs to its customers. For example, Snowflake expects the users to decide on the size and number of their virtual warehouses, clustering keys, and other parameters; Azure Synapse expects them to select compute pool sizes; Amazon Redshift expects users to decide on cluster size, node type, and Redshift Processing Units (Redshift Serverless). However, making optimal decisions for millions of daily queries that are issued at different times and by different teams and applications is a daunting task for a typical clouds user. Even when manual optimizations are possible, they have to be constantly redone since cloud workloads constantly change.

Keebo's vision. Our goal at Keebo is to make data clouds smarter with a concept we call *data learning*. That is, we learn from how users and applications interact with their data in the cloud and use our trained models to automate the tedious aspects of those interactions [7]. Examples of such tedious tasks include optimizing individual queries, accelerating BI dashboards, reducing the cloud bill, discovering data quality issues, detecting changes to the underlying data distribution, identifying the most likely causes driving a KPI change, and even enforcing data compliance rules. Keebo is a data learning platform comprised of a number of individual modules that each automate one of these tedious tasks for users of data clouds. For instance, one of Keebo's most popular modules is its Warehouse Optimization that automatically optimizes the customer's cloud data warehouses to ensure best performance while minimizing the overall cost. Likewise, Keebo's Query Acceleration module automatically optimizes queries to speed up BI dashboards without any manual efforts needed from the user [24]. In this paper, we will introduce Keebo's Warehouse Optimization, and defer our Query Acceleration to a companion paper. (The rest of the offerings are not generally available yet.)

Fully-automated warehouse optimization. Unlike traditional query optimization, which optimizes *one query at a time*, our goal is to optimize the *entire* warehouse based on (i) all the queries in the workload and (ii) the customer's cost and performance objectives. We refer to this problem as *warehouse optimization*, whereby the corresponding warehouse optimizer can be deployed and scaled independently, thus fitting nicely in the data cloud architecture. Depending on the particular warehouse product at hand, warehouse optimization decisions may include tuning the warehouse knobs, provisioning its resources (e.g., compute, memory), scaling it up or down, scaling it in or out, suspending or resuming [1] it, consolidating multiple warehouses into one, and load balancing decisions. While there have been numerous recommendation tools and tuning advisors for database administrators (DBAs) [16, 17, 20, 21, 48], to the best of our knowledge, Keebo's Warehouse Optimization (KWO) is the first fully-automated warehouse optimizer on the market. That is, users only need to specify their performance and cost requirements and Keebo automatically handles the rest. For example, KWO makes and applies automated decisions in real-time that minimize the overall compute bill while ensuring the customer's performance goals.

Architecture and workflow. Figure 1 shows our architecture. Keebo is comprised of several modules and components. The customer's admin specifies their desired cost and performance requirements through Keebo's *web portal*. The *data learning (DL)* platform continuously reads the telemetry metadata from the customer's CDW to train domain-specific *smart models* for each module (i.e., Warehouse Optimization, Query Acceleration, etc.). Here, we focus on Keebo's Warehouse Optimization (KWO) module. The DL platform trains a separate warehouse optimization model per each of the customer's warehouses to meet the unique performance requirements for the workloads observed on each warehouse. The smart models make real-time decisions, called *actions*, for each of the customer's warehouses, by taking four inputs into account: (1) the historical knowledge learned during the training phase, such as recurring patterns or query signatures and the impact of previous optimizations on their performance, (2) the *warehouse cost model*, which predicts the impact of each decision on cost and performance, (3) the customer *constraints*, which are a set of rules stating business requirements, such as 'prioritize performance over cost saving for this particular warehouse' or 'avoid downsizing between 9am to 9:30am', and (4) the real-time *feedback* from the *monitoring* component, such as increased latencies, new query pattern, queuing, or sudden spikes in the load. The smart models never take actions that violate the customer constraints. They rely on historical data to decide their optimization actions but they also back off and self-correct based on the real-time feedback they receive from the monitoring module. The actions are translated into the CDW vendor's own API and executed by the *actuator*. KWO offers a *value-based pricing*, which relies on the warehouse cost model to estimate the monetary savings brought to the customer

[1]Some CDWs, such as Snowflake and Azure Synapse, allow customers to suspend their warehouses that are not used temporarily in order to reduce costs.

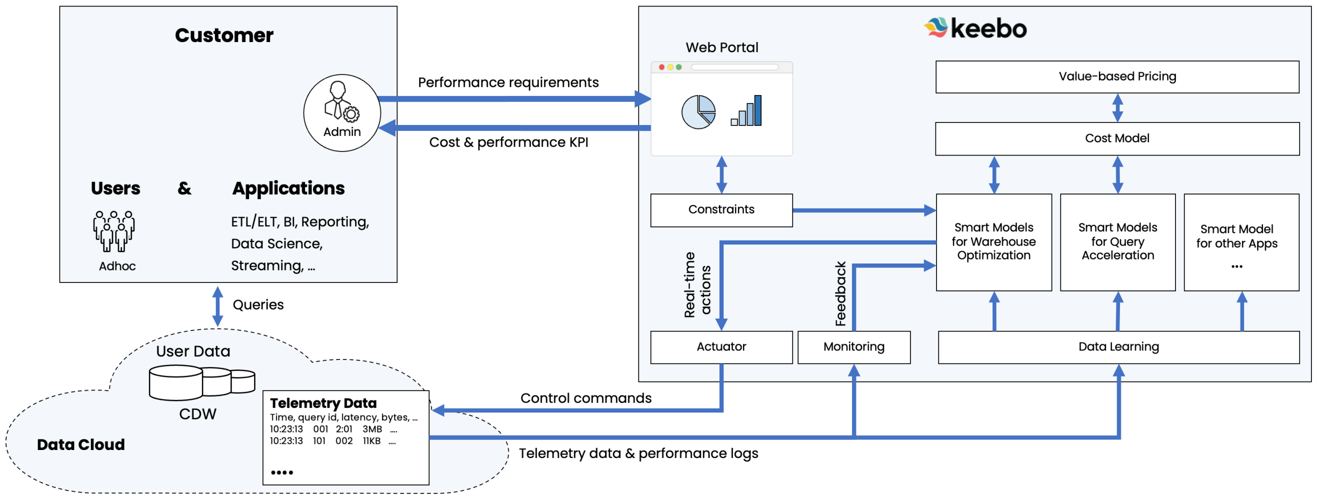

Customer
Admin
Performance requirements
Cost & performance KPI
Users
&
Applications
Adhoc
ETL/ELT, BI, Reporting, Data Science, Streaming, ...
Queries
User Data
CDW
Data Cloud
Telemetry Data
Time, query id, latency, bytes, ...
10:23:13 001 2:01 3MB ....
10:23:13 101 002 11KB ....
....
Control commands
Telemetry data & performance logs
keebo
Web Portal
Value-based Pricing
Cost Model
Constraints
Smart Models for Warehouse Optimization
Smart Models for Query Acceleration
Smart Model for other Apps ...
Real-time actions
Feedback
Actuator
Monitoring
Data Learning

Figure 1: Keebo's architecture.

as a direct result of KWO's own optimizations. The customer is charged a percentage of the actual savings. The customer can also inspect various cost and performance KPIs through the dashboards in the web portal.

Keebo's models constantly learn and improve with more usage. Depending on their workload, customers observe 20%–70% savings (i.e., reduction of their CDW bill). On average, customers reach 50%, 70%, and 95% of their eventual savings after only 20, 43, and 83 hours of onboarding, respectively.

To summarize, we make the following core contributions:

(1) We introduce a Data Learning approach to warehouse optimization, which—to the best of our knowledge—is the first fully-automated solution for optimizing cloud data warehousing and reducing costs.
(2) We present a value-based pricing strategy based on a novel warehouse cost model that quantifies the direct impact of various warehouse optimizations on the billable cost of a CDW. (Section 5)
(3) We present a detailed architecture for incorporating real-time performance feedback using deep reinforcement learning. (Section 6)
(4) We report empirical results on production workloads. (Section 7)

The rest of this paper is organized as follows. In Section 2, we present our design criteria and challenges. In Section 3, we provide an overview of various optimization decisions, followed by a description of our user flow and end-to-end architecture in Section 4. We introduce our novel warehouse cost model in Section 5, and our Deep Reinforcement Learning framework in Section 6. Finally, we present empirical results in Section 7 and overview the related work in Section 8.

2 DESIGN CRITERIA AND CHALLENGES

Below we present the main criteria and goals that have guided our design decisions (C1–C6).

C1. Zero barrier to adoption. One of KWO's overarching design goals is that it must not come with any downsides for the customer. In other words, to ensure fast, easy, and wide adoption, the customer should have nothing to lose in trying or buying KWO. This criterion has guided many of our other design choices: minimal on-boarding effort, little or no ongoing maintenance from customer, zero access to customer data, no slowdowns of customer's workload by default, value-based pricing (no savings, no charges), remaining transparent to users and applications, and the ability to accommodate various use cases and scenarios for arbitrary workloads.

C2. Ability to express customer's requirements. Data clouds serve a wide range of customers, from cost-conscious to performance-sensitive ones, from small data teams to large enterprises with tens of disparate teams. Even within the same organization, every application, workload and data warehouse has a different set of business requirements. A slowdown of an ETL job might cause SLA violations and immediate failure of upstream applications. Similarly, a slow BI dashboard can lead to user complaints, while a reporting application may be able to tolerate slightly longer query latencies. Therefore, one of our design goals is to allow customers to specify their business requirements. For example, a customer must be able to specify how aggressively or conservatively they want each of their warehouses to be optimized. As another example, they should be able to prevent KWO from downsizing their BI warehouse during mission-critical hours (e.g., Mondays 9-10am) or even request a larger warehouse during certain times of the day, week, or month.

C3. Fully autonomous. There have been numerous *recommendation* tools, whereby a set of suggestions are made to the DBA in terms of indexes, knobs values, settings, or other actions [16, 18, 21, 28, 29]. These tools have had limited adoption in the past, but are even less applicable to modern data clouds. Many customers are running millions of queries with significant fluctuations throughout the day. As a result, no static value will be optimal due to the unpredictable and time-varying nature of modern workloads. No data team will have sufficient engineering resources to continuously monitor their data cloud and manually apply performance recommendations throughout the day. Further, if the DBA could determine whether a particular recommendation provided by the tool will be effective, he or she would not need a recommendation tool in the first place. Often, given the sheer volume of queries and the ever-changing nature of cloud use cases, it is humanly impossible to determine an optimal course of action. One of design goals therefore is to ensure that KWO is fully-automated, i.e., rather than providing recommendations it can perform the optimal actions in real-time, learn and adjust on its own. That is, while the admin can intervene or perform manual action, the product should have the capability to be fully autonomous.

C4. Prioritizing performance over savings. In some cases, improving performance may reduce costs too. For instance, if doubling the warehouse doubles the hourly rate but allows the jobs to finish in one hour instead of three hours (e.g., by allowing the entire data to fit in memory), then the overall cost and performance have both improved. In other cases, there may be a trade-off between cost and performance. While KWO must allow users to make such tradeoffs (see C2), it must never prioritize cost over performance without customer's explicit request. In other words, when faced with a choice of saving costs with a moderate slowdown versus less (or no) savings but no slowdown, KWO's default behavior must be the latter. The reason is that most customers are less forgiving of performance slowdowns and user complaints than they are of lower savings, i.e., savings are always welcome even if modest.

C5. Workload agnostic. KWO must not make any *a priori* assumptions about the customer's workload. Even for the same customer, each warehouse might be drastically different. A warehouse serving an ETL workload might have a highly-recurring query pattern, while a warehouse serving data analysts might be serving ah-hoc queries with significantly larger load near the month end. In fact, the main reason the out-of-box performance of data clouds is not optimal for most customers is the general-purpose design of CDWs. In many cases, CDWs face hybrid or even homegrown and highly custom applications at each customer that they have not observed from other customers. Therefore, KWO must train a new smart model from scratch for each of the customer's warehouse based on the workload at hand.

C6. Security. Data warehouses are home to the most valuable digital assets of the company. As such, security concerns and scrutiny are heightened for products that access the customer's data warehouse. While access to query text and customer data is essential for Keebo's Query Acceleration, KWO must not store or even access any customer data. Specifically, KWO must train its smart models only using telemetry and performance metadata. Even query texts and usernames that are often present in performance must be securely hashed to ensure personally identifiable data are never leaked to KWO. This is critical to removing security concerns in adopting KWO (see C1).

3 OVERVIEW AND SCOPE

In this section, we provide a brief overview of various warehouse optimization actions that one can take. While each CDW vendor exposes a different set of knobs and levers, here we focus on Snowflake as one of the most popular CDW products on the market given it is also the most popular CDW among Keebo's customers. We defer warehouse optimization of Google's BigQuery to a separate paper. Even Snowflake itself offers a variety of warehouse optimization opportunities. However, due to space constraints, in this paper we limit our discussion to three main types of warehouse optimizations.

Memory optimization. Snowflake is capable of automatically suspending a warehouse when it is idle for a set period of time (a.k.a. *auto-suspend interval*) and resuming it automatically when the next query arrives [8]. Since Snowflake charges customers based on the number of minutes the warehouse is running, this feature frees customers from having to manually manage their warehouse schedule according to their query arrival times. However, the customer still has to choose a static value for their auto-suspend interval. Setting a small auto-suspend interval has drawbacks. This is because each time a warehouse is suspended, its local memory cache is dropped, causing future queries to read data from cold storage when the warehouse is resumed. Depending on the workload, this may have a dramatic impact on query latencies and thereby the total time the warehouse is running, which itself increases the overall cost. For example, queries in BI workloads tend to access similar data and therefore are more cache-sensitive. Frequent warehouse suspension and dropping of the cache, will impair the user experience but will also increase the execution time and the warehouse usage. Likewise, setting a large auto-suspend interval will also increase cost by causing the customer to pay for the idle warehouse time. There are several rules of thumb for setting the auto-suspend interval [2], but all of them require the customer to continuously monitor the workload and analyze the impact of losing the cache versus keeping the warehouse running. These rules of thumb provide no guarantees on optimal cost or performance.

Warehouse resizing. Snowflake requires users to select a size when creating a warehouse. The warehouse sizes come in abstract "T-shirt sizes" – X-Small, Small, Medium, Large to 6X-Large [14]. The costs double with each larger size, e.g., Small costs twice the X-Small, Medium costs twice the Small, and so on. The available compute resources for each size are not publicly documented due to differences in provisioning among cloud providers, but the compute capacity is widely assumed to also double with each increment in warehouse size [4]. Aside from a beta feature, Snowflake's generally available product does not offer any option to scale up or scale down the warehouse sizes automatically. However, the customer has the option to resize the warehouse manually. Snowflake recommends customers choose their warehouse size according to the complexity of their query workload [12]. Unfortunately, it is hard for customers to estimate the optimal resources for their queries. Instead, most customers resort to a crude strategy whereby they experiment with different warehouse sizes to find one that offers reasonable performance for their peak load. Since the load and queries vary throughout the day, a fixed size is almost never optimal. Unfortunately, even these crude experiments are only done occasionally (e.g., at provisioning time), and thus the selected warehouse can become even less effective as the workload evolves. Repeating these experiments frequntly is too tedious for most customers.

Warehouse parallelism. By default, Snowflake creates a single cluster for each warehouse. Each cluster can run multiple queries. However, if sufficient resources are unavailable on the cluster, the queries are queued till the resources become available. To reduce queuing delays, Snowflake offers a scale out capability called a multi-cluster warehouse [11], whereby new clusters are created on demand to provide additional resources. In multi-cluster warehouses, Snowflake can automatically add/remove clusters in response to changes in the workload. The clusters can increase up to the maximum cluster count specified by the customer. The Snowflake scheduler considers the queue sizes and available resources on the existing clusters before scaling out to the maximum cluster count.

Snowflake offers two dynamic scale-out policies—*Standard* and *Economy* [13]. The Standard policy prevents queuing by aggressively scaling out, while the Economy policy reduces cost by keeping the clusters fully occupied. Customers can also choose a static *Maximized* mode, where the minimum and the maximum number of clusters are equal, causing the warehouse to start and keep all the clusters concurrently. The *Maximized* mode can be an option for customers with non-fluctuating workloads or zero tolerance for cluster startup delays. Note that all clusters in a multi-cluster warehouse are of the same size. That is, a warehouse resize operation affects all the clusters in the warehouse. Customers can manually scale up via resizing and scale out via multi-cluster warehouse simultaneously, which increases the complexity of finding the optimal warehouse resource configuration for customer's workload.

4 END-TO-END ARCHITECTURE

KWO is a fully managed SaaS product that plugs into existing CDW infrastructure. That is, customers do not need to install or deploy any software. They can continue using their data clouds as before, while the managed service automatically identifies and performs warehouse optimizations in the background. Once connected, Keebo covers the entire optimization life-cycle of the data warehouse, from observing the workload, learning smart models, applying optimization decisions, monitoring the performance impact of those decisions, adjusting or reverting the optimizations in case of an adverse impact, and reporting the overall benefits to users. Figure 1 shows the architecture of Keebo's warehouse optimization. Next, we describe each component.

4.1 User Interface

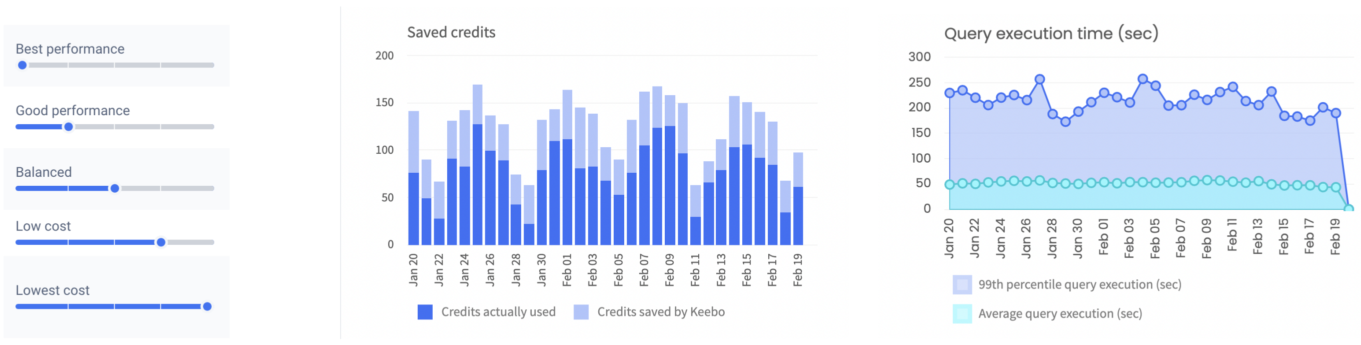
KWO offers an API service for programmatic access as well as a web interface. The majority of uses and applications using the CDW do not need to interact with or even be aware of Keebo. Only the customer's admins need to access Keebo to configure and monitor the service. The web portal is shared among Keebo's different applications (Warehouse Optimization, Query Acceleration, etc.), as customers may sign up for multiple applications. Here, we focus on the user interface for KWO, which offers three main functions: (i) performance dashboards (ii) sliders, and (iii) constraints.

Dashboards. The dashboards offer a comprehensive view of various KPIs, with the ability to filter by time and warehouse name, or aggregate daily, weekly or monthly. The KPIs include metrics such as the CDW spend, the savings brought by KWO, query latency and queue times (both average and 99th percentile), and cost per query. KWO also offers full visibility to customers by visualizing the real-time actions taken on each warehouse. Figure 2 (b)–(c) show screenshots of daily cost savings and query latencies.

Sliders. KWO provides a single *slider* per each warehouse, as shown in Figure 2 (a), with five positions ranging from "Best Performance" to "Lowest Cost". These positions allow the customer to decide its desired trade-off between performance and cost for that particular warehouse. That is, through this slider the customer controls whether KWO should lean towards cutting costs (i.e., be more aggressive) or preserving performance (i.e., be more conservative). By default, the sliders are in "Balanced" position, which means KWO will aim to only apply optimizations that cut cost without degrading performance. If the customer is willing to accept a small degradation of performance, they can move the slider to "Low Cost". Similarly, if the customer wants to further reduce the chances of any slowdown, say by provisioning for sudden spikes in usage, they can move the slider to "Good Performance" position. This customer is in charge of the slider position because only they know how mission critical each workload is.

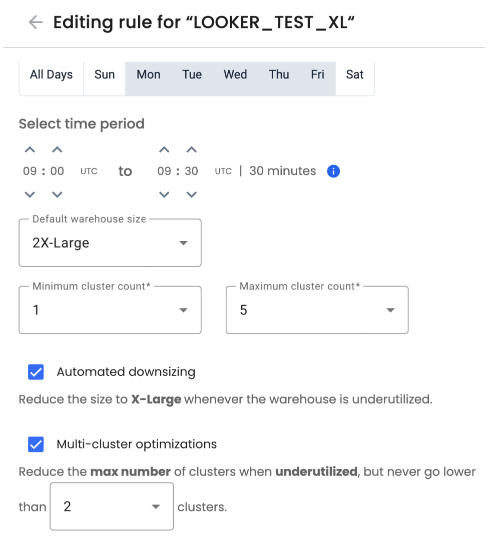
KWO performs different optimizations (see Section 3) that often interact and compete with one another in complex and non-linear ways. For example, the impact of downsizing a warehouse depends heavily on the extent to which we also reduce the degree of parallelism (e.g., number of clusters). A salient feature of KWO is that, instead of forcing customers to reason and worry about the impact of each optimization independently, they can simply use a single slider to express their high-level trade-off and rely on KWO to translate that into specific guidelines for each type of optimization underneath. In other words, KWO simplifies the tuning of the aggressiveness for various optimizations by unifying them into a single slider, and mapping it internally to various hyper-parameters of the learning algorithm that can deliver the customer's cost-performance trade-off.

Constraints. KWO's interface allows admins to also express their hard constraints through defining rules. Although KWO automatically adjusts its optimizations dynamically based on real-time changes of the workload and based on the trade-off slider, customers can also define additional rules for a more deterministic behavior. In each rule, the customers can disallow or allow certain optimizations or enforce certain resources during certain hours of the day or days of the week for each warehouse. For example, a rule may state that from 9am to 9:30am the BI warehouse must change from Large to X-Large with a minimum of 3 clusters, or that on last day of the month the warehouse used for adhoc analytics cannot be downsized even if underutilized, but KWO may still adjust its degree of concurrency. KWO's automated optimizations always respect the customer provided rules, treating them as hard business constraints. Figure 3 shows a screenshot of an example constraint.



(a) Sliders **(b) Performance: Savings** **(c) Performance: Latencies**

Figure 2: Keebo's web interface: settings and performance dashboards.



Editing rule for "LOOKER_TEST_XL"

All Days | Sun | Mon | Tue | Wed | Thu | Fri | Sat

Select time period

09 : 00 UTC to 09 : 30 UTC | 30 minutes

Default warehouse size: 2X-Large

Minimum cluster count*: 1

Maximum cluster count*: 5

☑ Automated downsizing

Reduce the size to **X-Large** whenever the warehouse is underutilized.

☑ Multi-cluster optimizations

Reduce the **max number** of clusters when **underutilized**, but never go lower than 2 clusters.

Figure 3: Example of a business constraint rule.

4.2 Data Learning

The core of Keebo's architecture is its *data learning* platform, which continuously trains domain-specific *smart models* for each application (Warehouse Optimization, Query Acceleration, etc.) based on how users and applications interact with the underlying data. Since our focus in this paper is KWO, here we only discuss how our data learning platform trains smart models for warehouse optimization. In this case, the training data is the performance telemetry metadata provided by the CDW, such as the number of queries, their arrival, queuing, and completion time, the number of bytes they each scanned, etc. In Section 6.1, we provide a detailed description of the metadata that we use for training KWO's smart models. Instead of training a single smart model per customer, we train a separate warehouse optimization model for each of the customer's warehouses in order to meet the unique performance requirements of the different workload observed on each warehouse. In Section 6, the algorithmic aspect of our data learning platform in more details. The most notable aspect of our data learning is that it generates models that continuously improve over time, i.e., deliver better performance and higher savings. This is because they constantly (i) train on additional data, and (ii) learn from how the past actions impacted cost and performance and thus auto correct. As a fully-managed service, the data learning process is completely automated and transparent to users. Moreover, for privacy and security reasons, the smart models trained based on a customer's metadata are never shared or used for other customers.

4.3 Smart Models

Each of the customer's warehouses is assigned a unique *smart model*,[2] which makes real-time decisions, called *actions*, for that particular warehouses. The smart models have been trained on past performance telemetry metadata during the training phase. Thus, they're aware of the past workload running on their corresponding warehouse. For example, the models capture whether the workload exhibits a recurring load during certain times or days, or if the queries in the workload share a similar pattern. If the workload is highly unpredictable, the models capture that too, i.e., the models are aware of their own confidence in predicting future workload. In the addition to the historical behavior, the smart models take three additional inputs into account when deciding on their next actions:

(1) the *cost model*, which predicts the impact of each decision on cost and performance (Section 4.6,
(2) the customer *constraints* and *slider* trade-off, which specify the business requirements (Section 4.1), and
(3) the real-time *feedback* from the *monitoring* component, such as increased latencies, new query pattern, queuing, or sudden spikes in the load (Section 4.4).

The smart model may adjust its action based on the latest slider position. For example, if the customer changes their slider position from 'Balanced' to 'Good Performance', there is no need for retraining the smart model from scratch. Instead, the smart model can re-calibrate its decisions automatically. However, the smart model never takes actions that violate the customer constraints. For example, if the customer has recently banned downsizing during a time-window, non-compliant actions are cancelled and replaced

[2]We use the term 'smart model' to differentiate it from traditional models: as our algorithms consult other components at run-time (e.g., cost model and constraints) rather than blindly applying the actions suggested by the original model that was trained offline.

with the next best action that complies with the latest constraints. Finally, depending on the real-time feedback, the smart model may back off and rolls back the previous settings of the warehouse. For example, if the monitoring data indicates a sudden spike in the load or additional queuing in the warehouse, the smart model will immediately self correct and resort to a more conservative action. The final action is sent to the *actuator*, which we describe in Section 4.5.

4.4 Monitoring

KWO continuously monitors the customer's warehouses for three reasons. First, it monitors performance-related metrics (such as query latency and queue times, queue sizes, etc.) in real-time to assess the impact of its own actions and provide feedback to the smart models so they can self-correct and adjust their next actions accordingly (Algorithm 1, lines 18–19). Second, since models have been trained on past telemetry data, it monitors for sudden spikes and changes in the workload to back off from previous actions if needed. Finally, it monitors whether external applications have made any modifications to existing warehouses. Particularly in larger organizations, it is always possible that users may directly interface with the CDW and make modifications that conflict with KWO's actions. For example, KWO may decide that downsizing a warehouse but increasing its cluster count will lead to cost savings without negatively impacting performance. However, if another user simultaneously decides to reduce the cluster count, the end result can be devastating for performance. Therefore, as soon as KWO detects external changes made to a warehouse, it immediately reverts its own action and only resumes its optimizations whenever (i) the external changes have been undone, or (ii) the admin explicitly asks the optimizations to continue.

4.5 Actuator

The actuator is responsible for translating the actions decided by the smart model into the API of the underlying CDW and execute them. The actuator therefore serves as a layer of abstraction between Keebo and the underlying CDW, hiding the vendor-specific implementation details from the smart models. It keeps a record of all actions taken and reports any errors it encounters. In the specific case of Snowflake, most actions turn into `ALTER WAREHOUSE` commands. For example, the following query changes a warehouse called "COMPUTE_WH" to Medium size:

```
ALTER WAREHOUSE COMPUTE_WH
SET WAREHOUSE_SIZE=MEDIUM;
```

4.6 Warehouse Cost Model

KWO relies on a warehouse cost model to estimate the savings realized by customers as a result of KWO's direct actions and optimizations. To achieve this goal, the cost model has to estimate the billable cost of the CDW with and without Keebo's warehouse optimizations. This is in contrast to traditional cost models, which aim to produce an abstract number for comparing pairs of query plans. Our warehouse cost model allows the smart models to make informed decisions when deciding their next action (see Section 4.3). It also allows KWO to be offered with value-based pricing, which we discuss next. We present our cost model in more details in Section 5.

4.7 Value-based Pricing

As a fully managed warehouse optimization service, KWO is priced based on the value it produces. That is, customers are charged a percentage of the actual savings realized as a direct result of KWO's actions. The customer can also inspect various cost and performance KPIs through the dashboards in the web-portal.

With a value-based pricing, there is no lock-in or upfront cost. Instead, customers only pay for the value already delivered to them. This pay-as-you-save model aligns nicely with the pay-as-you-go model offered by most data clouds. As previously mentioned, we rely on our warehouse cost model to estimate the overall CDW costs had the customer not had KWO in place.

5 WAREHOUSE COST MODEL

Goal. The goal of Keebo's warehouse cost model is to estimate the billable cost of the warehouse with and without Keebo's warehouse optimizations, and thereby estimate the savings resulted from Keebo's actions brought to the customer. This is important for two reasons. First, it allows KWO's smart model to make more informed decisions when deciding its next action (see Section 4.3). Second, it helps customer view the quantitative impact of KWO on reducing their overall CDW bill, which can then be used for a value-based pricing (see Section 4.7). For example, assume KWO performs a certain number of optimizations between 9am and 10am. The customer can later see the bill from their CDW vendor for that hour, but they will not know how much they *would have paid* had they not have Keebo's optimizations in place. That is, the customer would not know if KWO reduced or increased their spend (and if so, by how much) simply from their CDW bill. To solve this problem and offer assurance and visibility to the customer, Keebo's warehouse cost model uses a *what-if* analysis to estimate what the cost would have been had KWO had not performed its optimizations.

Before presenting how our warehouse cost model works, it is worth noting how our proposed warehouse cost model differs from (and is more challenging to develop than) cost models used in traditional query optimization.

Differences from traditional cost models. Unlike traditional query optimization cost models that produce an abstract metric which could only be used for comparing two plans, the warehouse optimization cost model directly estimates the billable cost incurred by the CDW (e.g., credits for Snowflake, bytes scanned for BigQuery, and hours of usage for Azure Synapse). While developing a cost model is always challenging, developing a warehouse cost model is even more so for a number of reasons.

First, the warehouse cost model must estimate the cost of the entire warehouse, rather than the cost of each individual query. Instead, we need to capture all the application and user interactions using the warehouse (e.g., query arrival and completion times, periods of no activity, etc.) and compute an aggregated measure.

Second, the warehouse cost model must estimate the cost of the warehouse using absolute values that map directly to the dollar amount billed by the CDW vendor. For example, for Snowflake, estimating the number of credits consumed by a warehouse can be mapped to a dollar amount based on Snowflake's cost per credit negotiated by the customer. This is in contrast to traditional cost models that emit abstract and unitless metrics, which do have any meaning and can only be used for comparing various query plans.

Third, instead of comparing query plans, the warehouse cost model is used to compare the optimized warehouse state with its original state. We refer to these two states as *with-Keebo* and *without-Keebo*, respectively. Thus, unlike traditional cost models that rely on a compile-time search, our warehouse cost model relies on 'what-if' analysis. That is, it measures how far the cost of the with-Keebo state is from that of the without-Keebo one, and emits the difference as cost savings presented to the customer. This also means the warehouse cost model has to be more accurate than trantional cost models, particularly when used as a basis for pricing (see Section 4.7).

Our approach. Our approach to overcoming these challenges is to combine analytical and machine learning models. Our analytical models capture the underlying CDW's behavior at the fine-grained query level using certain parameters, while our machine learning models calibrate and estimate those parameters based on macro trends in the customer's own historical data. The advantages of using this hybrid learning approach are that (i) it gracefully adapts to the workload at hand, (ii) it is robust against both micro and macro trends, (iii) it is easily extensible to new CDW products, and (iv) it constantly improves over time as it observes more data.

We next describe the two major components of our warehouse cost model: analytical query replay (that captures per-query system behavior) and machine learning-based parameter estimation.

5.1 Query Replay

The first step in our warehouse cost estimation is to run a *what-if analysis* by *conceptually* replaying the queries in the workload and estimating the without-Keebo costs, i.e., if none of the currently applied warehouse optimizations were in place. In most cases, however, the with-Keebo cost need not be estimated as it can be directly obtained from the CDW's billing data for the period that KWO was actively optimizing the customer's warehouse. The difference between the estimated without-Keebo cost and the actual with-Keebo cost is KWO's cost saving delivered to the customer.

Using Snowflake as our running example, since they bill hourly for per-second usage in that hour, our query replay iterates over all queries run during each hour and computes the number of seconds the warehouse was active in that hour. To compute the active seconds, the query replay considers periods of time in which at least one query was running, inclusive of idle times (such as warehouse startup, shutdown) and the auto-suspend interval before an idle warehouse starts to shutdown. For multi-cluster warehouses, the query replay also considers how many clusters would be active at each point in time and multiplies the active seconds accordingly. The query replay also considers the warehouse's original cluster scaling policy[3] that was in effect at that time to estimate how soon new clusters would have been added or removed had Keebo's optimizations not been in place.

At each step during the replay, we consider the customer's original settings without any of Keebo's warehouse optimizations. For example, for any period that KWO has changed the warehouse size, we consider the warehouse's original size (and hence hourly rate) for estimating the without-Keebo cost. Likewise, if Keebo suspends a warehouse sooner than the customer's original auto-suspend period, we use the the latter, and so on. Finally, we multiply the total active seconds by the hourly rate of the original warehouse size to estimate the without-Keebo cost.

The query replay grounds our cost estimation into the actual query processing and billing behavior of the underlying warehouse, thereby scaling gracefully as workloads change over time. For instance, if there is a sudden drop in the number of queries then the estimated without-Keebo cost will also adjust automatically. Likewise, different scaling policies or new warehouse optimizations can be incorporated into the query replay. Overall, query replay prevents our cost model from being a black box; rather it makes cost and saving estimates more explainable and transparent to the customer.

5.2 Parameter Estimation

The query replay simulates how each query would have behaved in the absence of Keebo's optimizations. However, in doing so, it requires certain parameters that may vary for different workloads and warehouses. Next, we discuss how we use machine learning models to estimate these parameters from historical data for the warehouse at hand.

Impact on query latencies. The query latency often varies with the warehouse resources, and therefore the same query may exhibit different latencies when run on different warehouse sizes. For example, if KWO changes the size of a warehouse, we cannot later assume the queries would have had the same latency had Keebo not changed the size. To estimate the impact of warehouse size on query latencies, we train a regression model to scale query latencies across warehouse sizes. Fortunately, since KWO changes warehouse sizes dynamically, it is likely to find identical or at least similar queries[4] run on different warehouse sizes over time. In situations where we do not find similar queries in the past, we use the average impact on query latencies observed on that warehouse as a first-order approximation. We use all such past observations as training data. Then, during query replay, we scale the execution time of each query according to the historical behavior we have observed for that query using our regression model.

It is worth noting that halving a warehouse size may or may not reduce cost. This is because the latency may grow super-linearly for some queries, but linearly or sub-linearly for others. For example, in the extreme case, if all queries more than double in latency, then the downsizing will indeed increase the overall costs. However, in general, for usage-based CDWs such as Snowflake, slowing down a subset of the queries may not increase the overall cost, depending on their arrival times. For example, one query's execution period may be subsumed within that of another query.

Impact on query arrival times. Impacting query latencies will also impact the gap between their arrival times. For example, if KWO had not increased the warehouse size, we might have had

[3]For example, Snowflake customers choose between standard and economy policies.

[4]Note that KWO does not have access to plain query texts for security reasons (Section 2). Thus, we use the hash value of the query text and the hash value of the query template (i.e., query text stripped of all constants) to find identical and similar queries, respectively.

longer query execution times, but also smaller gaps between subsequent queries if we assume their arrival times would remain the same. These changes in query gaps, however, are artificial since in practice queries either arrive independently at a given arrival rate or they have dependencies that cause them to arrive at successive or scheduled time periods. Regardless, the gaps between should not change with warehouse optimization, even if the start and end time of each query changes. To account for this, we model the query gap intervals for each warehouse over time and use that model to predict and adjust the gaps during query replay. This process also accounts for the fact that query gaps cannot be longer than the auto-suspend interval since the warehouse would have shut down after that period of inactivity, causing the warehouse costs to stop anyways.

Impact on warehouse parallelism. Finally, the arrival pattern of queries will impact the number of clusters that would have been spun up without Keebo's optimizations. For example, if KWO limited the maximum number of clusters to 4 while the original value was 10, then the with-Keebo scenario could use between 1–4 clusters whereas the without-Keebo scenario could have used anywhere between 1–10 clusters. We therefore need to predict the number of clusters that would have been used at each point in time in order to estimate the impact of KWO's actions. We train cluster-count predictor using the past performance statistics and the original max cluster count. To avoid dealing with per-second predictions, we batch the past query execution into mini-windows and then predict the average cluster count for each mini-window. Once we have the predicted number of concurrent clusters, we scale the costs accordingly during the query replay.

Calibrating the parameters used during the query replay with learning-based models makes our warehouse cost estimator resilient to simulation errors, yielding more accurate estimates (see Section 7.2).

6 DATA LEARNING

6.1 Training Data

KWO relies on CDW's performance telemetry to learn query and usage patterns of each warehouse and use the resulting smart models to guide its optimizations. KWO only uses warehouse and query metadata information for training its models. For security reasons, It does not use the text of the queries or the customer's data used in those queries. This metadata is fetched periodically (Algorithm 1, line 14).

The metadata used in training comes from two sources: query history and billing history. The query history includes system information (e.g., warehouse name, warehouse size, cluster number), timeseries data (e.g., query arrival times), performance metrics (e.g., query latency, queuing delays, bytes scanned), and so on. The billing history includes information about the costs incurred by the CDW, such as date, time and usage. Next, we explain how our data learning platform uses this training data.

6.2 Memory Optimization

As mentioned in Section 3, usage-based CDWs allow users to suspend and resume their warehouses as needed. However, the decision

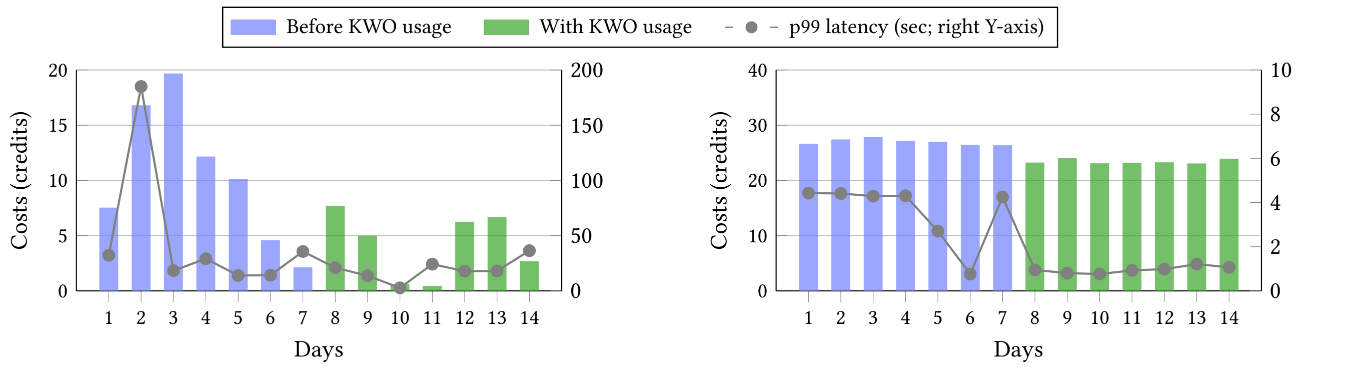
Algorithm 1 Data Learning Algorithm

```
1:  Input
2:  aggr : The slider position (i.e., aggressiveness)
3:  wh : warehouse name
4:  WCM: warehouse cost model trained for wh
5:  UC : user constraints
6:  T : frequency of collecting detailed telemetry data
7:  $T_{realtime}$ : frequency of checking real-time performance
8:  Initialization
9:  D ← ReadTelemetryData(last 90 days)
10: action[0] ← null, reward[0] ← null
11: i ← i null
12: while true do
13:     if T hours have elapsed since last training then
14:         D ← D ⋃ ReadTelemetryData(last T hours)
15:         M ← TrainSmartModel(D, wh, aggr, WCM)
16:     end if
17:     if $T_{realtime}$ mins have elapsed since last action then
18:         feedback[i-1] ← Monitoring.RealTimeState()
19:         action[i] ← M.nextAction(UC, WCM, feedback[i-1])
20:         Actuator.apply(wh, action[i])
21:         i ← i + 1
22:     end if
23:     savings ← cm.estimateSavings(action[], feedback[])
24:     report(action[], feedback[], savings)
25: end while
```

about when to suspend and resume, in a way that achieves both high performance and cost efficiency, is non-trivial for most real-life workloads. This is because (1) computing the decision's impact can be complex, (ii) workloads may change over time, and (iii) there can always be ad-hoc queries that have not been observed previously. Keebo takes a learning-based approach to solve this problem.

Tradeoff. Before describing our approach, we discuss the tradeoff in suspending a warehouse. On the one hand, suspending a warehouse for the next T seconds can be beneficial because, if no queries arrive in that period, we can simply save warehouse costs. On the other hand, the penalty of suspending a warehouse can be greater than the benefit if queries appear unexpectedly during the period we expected no queries to appear. This negatively impacts query latencies because a restarted warehouse is less like to have relevant data in its local memory. That is, we can get more benefits from suspending a warehouse when the warehouse is likely to be idle.

Learning-based approach. Ultimately, we need to decide if it is worth suspending a warehouse w *right now*. Let $(1 - \delta_T)$ be the chance that there will be no queries (submitted by users) for the next T seconds. Then, the expected cost savings from suspending the warehouse *now* will be $U_w \times T \times (1 - \delta_T)$, where U_w is the unit cost of the warehouse w. The penalty of suspending a warehouse is the slowdown experienced by the subsequent queries until the local cache is warmed up. We use d (seconds) to collectively quantify the total slowdown experienced by all such queries; then, this value is multiplied by the unit warehouse cost U_w to obtain an overall monetary penalty; the values of d is estimated from historical telemetry



(a) Savings for WarehouseA (less predictable workloads) **(b) Savings for WarehouseB (more predictable workloads)**

Figure 4: Keebo reduces warehouse costs with minimal impact on query performance.

data. After all, KWO suspends a warehouse if the following holds:

$$U_w \times T \times (1 - \delta_T) \; - \; U_w \times d \; > \; \varepsilon$$

where ε is a margin of error determined based on the slider position selected by the user (i.e., aggressiveness). T, δ_T, and d are all (periodically) learned from warehouse-specific telemetry data discussed in Section 6.1.

6.3 Deep Reinforcement Learning

At the core of KWO's learning is a variant of deep reinforcement learning (DRL) [19], designed to (1) decide on workload-specific warehouse optimizations and (2) quickly react to unexpected changes in the workload. In this section, we describe how we use and extend the DRL framework for our warehouse optimization problem.

In DRL, the model takes an action in an environment and receives feedback in the form of the new *state* of the environment and its associated reward. Based on this feedback, the model adjusts its internal weights and decides its next action accordingly.

As shown in lines 13 to 16 of Algorithm 1, we periodically retrain our neural network using the last 90 days of training data (D) in a supervised manner. We also augment our training data with the estimated impact of various actions using our warehouse cost model (WCM). We also use the slider position of the given warehouse (aggr) in our reward function to achieve the user's desired balance between cost and performance during training.

Before taking each action, the current state of the environment in terms of real-time performance metrics is collected from the monitoring component. The smart model takes the latest state as feedback and uses it internally to compute a numeric *reward* value for each possible action. The reward is a function of various system parameters, such as query latencies (L), query queue time (QueueT), and the warehouse bill (B), and is used to inform the model's decision-making process and improve its actions over time. The smart model also invokes the warehouse cost model and picks an action that maximizes the reward while satisfying user constraints (UC).

Our reward function combines various factors, as follows:

$$R = f(L,\ QueueT,\ B; \quad W_L,\ W_Q,\ W_B; \alpha) \qquad (1)$$

where W_l, W_q, and W_c represent the weights assigned to L, QueueT, and B, respectively. This reward function also has a parameter α that determines the balance between performance and costs. Adjusting the value of α allows the algorithm to prioritize actions that reduce the chances of query slowdown at the expense of lower savings.

We model the warehouse optimization problem as a Markov Decision Process (MDP) [43], and utilize a *Deep Q-learning* approach [27] to determine the optimal action. Our goal is learn a strategy that maximizes the total quality of our actions. This is a continuous learning process, whereby our model model collects more feedback on its actions in terms of their impact on performance and cost over time. This adaptation allows our model to automatically adjust to dynamic workloads.

In Q-Learning, the effect of each action is represented using Q-values $Q(s[i])$, where $s[i]$ is the state (i.e., feedback) received in i-th iteration. The Q-value is calculated by adding the maximum reward attainable from future states to the reward for achieving the current state. Formally, after each action action[i] on state $s[i-1]$, our model learns by updating its strategy $Q^{new}(s[i])$ using:

$$Q\,(s[i-1]) + \alpha\,[R_{i-1} + \gamma \max \alpha\,(Q\,(s[i], \alpha) - Q\,(s[i-1]))]$$

where γ and α are standard terms, called the discount factor and the learning rate, respectively. γ weighs the contribution of short-term and long-term rewards. α determines to what extent newly acquired information overrides old information. As the model gains more experience from optimizing the warehouse and receiving feedback, the Q-values converge to the optimal policy [34].

7 EMPIRICAL RESULTS

In this section, we present several experiments to show:

(1) Our automated warehouse optimization can reduce warehousing costs significantly after a short period of time (section 7.1).
(2) Our warehouse cost model yields highly accurate estimates of actual warehouse costs (section 7.2).
(3) Our optimizations incur almost no overhead (section 7.3).
(4) Our DLR-based approach allows users to achieve their desired tradeoff between cost savings and performance (section 7.4).

To measure *costs*, we use an abstract unit of *credit* because the dollar amount may differ from one customer to another depending on their business agreements with warehouse vendors.

7.1 Keebo Offers Significant Savings

To demonstrate the benefits of KWO, this section shares our empirical observations with production environments processing actual customer workloads. Specifically, we examine how the warehouses' credit usage right before and after using KWO. We intentionally choose a short period to reduce the chances of a significant change in the workload, thereby allowing us to use the pre-Keebo usage as a fair baseline to evaluate the with-Keebo behavior.

Cost Savings. Figure 4 shows the actual credit usage for two different warehouses (from different customers). In each subfigure, bars report credit usage, and a line reports daily 99th-percentile latencies. We first focus on the bars to compare credit usage before and with Keebo, which are distinguished using two different colors, blue and green, respectively. That is, in both subfigures, Days 8–14 are the days when KWO is enabled.

Figure 4a demonstrates savings for the warehouse with less predictable workloads; thus, its credit usage, even before KWO is enabled, fluctuates more than other warehouses we observe. Nevertheless, we observe that the overall credit usage becomes lower for the period when KWO is active (i.e., green bars). Specifically, the average daily usage reduces from 10.4 to 4.2 per day, meaning a 59.7% reduction. While these daily aggregates are useful for checking early results, our dashboard also offers weekly and monthly statistics for the users more interested in long-term behaviors.

In contrast, Figure 4b shows the warehouse data handling more predictable workloads, being indicated by relatively constant credit usage even before KWO (blue bars). With KWO, we observe that daily credit usage reduces by 13.2% from 26.9 to 23.4 per day.

Performance Impact. One of KWO's main goals is to avoid any negative impact on performance. To trace performance impact, we report 99th-percentile (p99) latency as the primary metric while our dashboards report other aggregate statistics as well. In Figure 4, the p99 latencies are depicted with lines. In these cases, we observe no noticeable latency changes with the introduction of KWO. In Figure 4b, p99 latencies are interestingly lower with KWO than before. Situations like this happen when KWO prefers constantly running smaller warehouses over sporadically running bigger warehouses in order to avoid cold cache and warehouse wake-up delays

7.2 Warehouse Cost Model is Accurate

Our warehouse cost model (Section 5) plays an important role for evaluating the savings KWO delivers to customers, and also for training our reinforcement learning models (Section 6.3). In this section, we evaluate the quality of our warehouse cost model using real-world query workloads sampled across several different warehouses.

In order to evaluate its quality, we examine how accurately our warehouse cost model can estimate the actual costs (reported by CDW) *without running any queries*. Then, these estimated costs are compared to the actual costs, which we obtain by actually running those queries. This task is more challenging than, for example, estimating how long it would take to completely process a set of queries by running them one by one, because the sampled workloads contain when those queries are submitted. That is, our warehouse cost model must incorporate not only the latencies of individual queries, but also, potential idle times between active query processing and its impacts on dollar bills, in consideration of warehouse-specific configurations (e.g., sizes, # of clusters, auto-suspend behaviors).

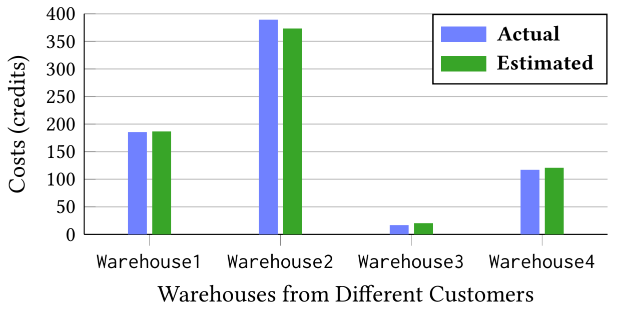


Figure 5: Our warehouse cost model offers highly accurate estimates of actual costs *without running any actual queries.*

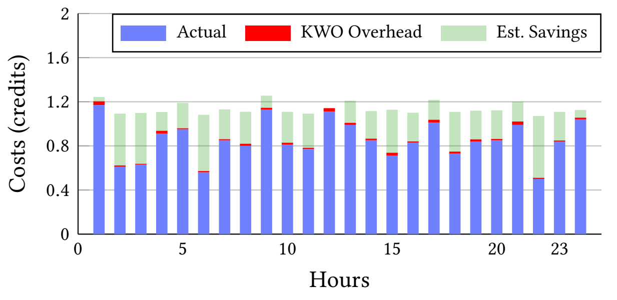


Figure 6: KWO incurs marginal overhead (red) compared to its savings (green).

Figure 5 shows our experimental results. In the figure, blue bars show the actual costs we have to pay to CDW for actually running queries; green bars show the estimated costs by our warehouse cost model, *without running any queries*. Intuitively, if the heights of those bars belonging to the same warehouse (e.g., Warehouse1) are identical, we can say the warehouse cost model is producing accurate estimates. For the workloads we consider, our estimates are nearly identical to the actual charges. Specifically, the relative errors are 0.67%, 4.09%, 20.9%, 3.12%, respectively. The relative error is greater for Warehouse3 because its absolute spending is quite low; even a small absolute difference makes the relative error large. Indeed, this is observed often for low-spending warehouses because even in production environments, there are provisioned, but rarely used ones. However, this is not a major concern for our warehouse cost model because such warehouses are anyway not the major target for optimization, with less opportunity for cost savings.

7.3 Keebo Incurs Almost No Overheads

KWO constantly monitors the amount of its overhead for obtaining telemetry data and performing actuator operations, because performing these operations also incurs (small) CDW costs. Our internal logic merges all the related data — the telemetry data for user queries, our overhead, and expected savings we offer — into one place, allowing our data scientists and automated monitoring tools to determine the final value provided by our service.

Figure 6 shows one such example. This figure shows hourly data of (1) actual credit usage (in blue), (2) KWO overhead (in red), and (3) estimated savings (in green). We observe several interesting trends here. First, KWO's overhead (in red) is negligibly small in comparison to other regular query processing. This low overhead is indeed the outcome of significant engineering efforts to (1) leverage running warehouses for obtaining telemetry data (without waking them), and (2) combine multiple queries into one to reduce the total runtime. Compared to overhead, estimated savings is significantly greater in amount, justifying the value of KWO for this warehouse. Second, the expected total spend without Keebo — the sum of Actual and Estimated Savings — are nearly identical over different hours. This is because the particular warehouse has relatively static workloads over time (for performing ETL tasks); yet, our optimization finds cost-saving opportunities.

7.4 Keebo Offers Intuitive Configuration Sliders

We have designed our algorithms with the ability to strike different tradeoffs between cost savings and query performance according to the customer's needs. This section confirms this behavior by running the same query workloads with five different configurations (i.e., from **Lowest cost** to **Best performance**) offered by our slider interface. Figure 7 shows the differences in warehouse costs (blue bars) and average query latencies (green line) according to the slider values. While this tradeoff relationship itself may not be surprising, what is meaningful about KWO is that *it is offering Pareto efficiency in managing warehouses.* In other words, for achieving average latency 1.42 seconds (slider value 3), KWO optimizes warehouse sizing and its operations so that the cost of processing a given workload is minimized.

8 RELATED WORK

There are three categories of prior work that are closely related:
Configuration. There is much work on tuning database configurations, either manually by database administrators or automatically via the use of machine learning as in OtterTune [46] and KEA [49]. Unfortunately, prior configuration tuning approaches do not consider the characteristics of the query workloads and rather tune based on system/machine level metrics. Furthermore, the tuning is not managed, i.e, they do not automate the end-to-end lifecycle.
Resource optimization. Recent work has explored the problem of optimizing resources, especially for analytical data processing systems in the cloud setting. Examples include Peregrine [25], Sparklens [15], and Seagull [39]. These approaches apply predictive, reactive, or adaptive techniques to find better resource allocation decisions for the data processing system. Such automated decisions, however, are not monitored or rolled back in case of performance degradation. They are also often employed for reducing vendor's operating costs, and may not be passed on to the customers.
Cloud WH Cost Optimizations. Several modern cloud data warehouses provide on-demand and easier-to-manage infrastructure,

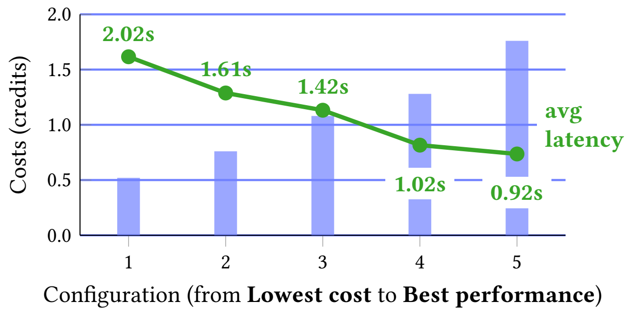


Figure 7: Keebo allows users to easily explore the tradeoff between cost savings and performance, with a single slider.

with features such as auto-scale [6, 13] and auto-suspend [5, 8]. However, they still need to be configured and evolved over time by the users. Other recommendation-based services such as Slingshot [10] or Bluesky Data [9] provide suggestions on how to tune such configurations with the users still responsible for applying and monitoring them over time. It is non-trivial for users to make price-performance trade-offs as they need to understand their workloads deeply and make decisions based on calculated judgment.
Reinforcement learning The past decade has seen wide adoption of machine learning (ML) in the database community. Recently reinforcement learning has been used across a variety of systems components. Query optimization [26, 32, 37] applied deep reinforcement learning to formulate better join ordering and eventually find better plans. However, the lack of sufficient training data requires the model to continuously update and improve its predictions as it learns from new examples. This process of online training can be computationally expensive and time-consuming. our DRL model benefits from having access to large historical telemetry data, which enables it to learn from a diverse range of past experiences without the need for constant updates. Reinforcement learning techniques have also been applied to managing elastic clusters[30, 36], adaptive query processing [44, 45], scheduling[31, 47], physical schema design [38], and tuning transactional databases [46].

9 CONCLUSION

Keebo is a data learning platform that trains models based on how users and applications interact with their data cloud, and uses those models to automate the tedious aspects of those interactions. In this paper, we introduced Keebo's Warehouse Optimization, as the first fully-automated solution for minimizing the cost of cloud data warehouses, while meeting the users' performance goals. Keebo seamlessly plugs into customer's existing data clouds and manages the entire optimization life-cycle of their CDWs. It automatically observes the workload, learns smart models, makes optimization decisions and takes actions in real-time, monitors the performance impact of those actions, adjusts or reverts its optimizations in case of an adverse impact, and estimates the overall savings brought to the customer as a result of those optimizations.

Keebo's models constantly learn and improve over time. Customers observe 20%–70% reduction in their CDW bill, and do so quickly: on average, they reach 50%, 70%, and 95% of their eventual savings after only 20, 43, and 83 hours of using Keebo.

REFERENCES

[1] Why is snowflake automatic clustering so expensive?, Oct 2019.
[2] 10 best practices every snowflake admin can do to optimize resources, 2020.
[3] Blog - for data team success, what you do is less important than how you do it, Feb 2021.
[4] Choosing the right warehouse size in snowflake, 2022.
[5] How to pause and resume dedicated sql pools with synapse pipelines, 2022.
[6] Using elastic resize in amazon redshift, 2022.
[7] Feb 2023.
[8] Automating warehouse suspension, 2023.
[9] Bluesky, 2023.
[10] Capital one slingshot, 2023.
[11] Multi-cluster warehouses, 2023.
[12] Selecting an initial warehouse size, 2023.
[13] Setting the scaling policy for a multi-cluster warehouse, 2023.
[14] Warehouse size, 2023.
[15] Welcome to sparklens report, 2023.
[16] Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, and Vivek R Narasayya. Autoadmin: Self-tuning database systemstechnology. *IEEE Data Eng. Bull.*, 29(3):7–15, 2006.
[17] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Materialized view and index selection tool for microsoft sql server 2000. *ACM SIGMOD Record*, 30(2):608, 2001.
[18] Ioannis Alagiannis, Debabrata Dash, Karl Schnaitter, Anastasia Ailamaki, and Neoklis Polyzotis. An automated, yet interactive and portable db designer. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1183–1186, 2010.
[19] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *IEEE Signal Processing*, 2017.
[20] Douglas P Brown, Jeetendra Chaware, and Manjula Koppuravuri. Index selection in a database system, March 3 2009. US Patent 7,499,907.
[21] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*, pages 3–14, 2007.
[22] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016.
[23] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923, 2015.
[24] Alekh Jindal, Barzan Mozafari, Yongjoo Park, Brian Westphal, Shi Qiao, Matthew Larsen, and Advait Abhay Dixit. Platform agnostic query acceleration, January 31 2023. US Patent 11,567,936.
[25] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. Peregrine: Workload optimization for cloud query engines. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 416–427, 2019.
[26] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.
[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
[28] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigumus, Junichi Tatemura, Neoklis Polyzotis, and Michael J Carey. Opportunistic physical design for big data analytics. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 851–862, 2014.
[29] Cristina Maier, Debabrata Dash, Ioannis Alagiannis, Anastasia Ailamaki, and Thomas Heinis. Parinda: an interactive physical designer for postgresql. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 701–704, 2010.
[30] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, et al. Park: An open platform for learning-augmented computer systems. *Advances in Neural Information Processing Systems*, 32, 2019.
[31] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*, pages 270–288. 2019.
[32] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proceedings of the VLDB Endowment*, 2019.
[33] Matillion. Survey: Rapid time to value is the main driver for analytics projects, yet making data available for insights is a barrier for 90% of enterprises, Oct 2019.
[34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.
[35] Siddartha Naidu and Jordan Tigani. *Google BigQuery Analytics*. John Wiley & Sons, 2014.
[36] Jennifer Ortiz. Perfenforce overview: A dynamic scaling engine for analytics with performance guarantees. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 31–33, 2017.
[37] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, pages 1–4, 2018.
[38] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-driving database management systems. In *CIDR*, volume 4, page 1, 2017.
[39] Olga Poppe, Tayo Amuneke, Dalitso Banda, Aritra De, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Deepak Shankargouda, Meina Wang, et al. Seagull: an infrastructure for load prediction and optimized resource allocation. *Proceedings of the VLDB Endowment*, 14(2):154–162, 2020.
[40] Research and Markets. Big data and analytics global market report 2022, by analytics tools, deployment mode, end use industry, application.
[41] Stas Sajin. Why is snowflake so expensive?, Sep 2022.
[42] Bhadresh Shiyal. Introduction to azure synapse analytics. In *Beginning Azure Synapse Analytics: Transition from Data Warehouse to Data Lakehouse*, pages 49–68. Springer, 2021.
[43] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
[44] Immanuel Trummer, Samuel Moseley, Deepak Maram, Saehan Jo, and Joseph Antonakakis. Skinnerdb: regret-bounded query evaluation via reinforcement learning. *Proceedings of the VLDB Endowment*, 11(12):2074–2077, 2018.
[45] Kostas Tzoumas, Timos Sellis, and Christian S Jensen. A reinforcement learning approach for adaptive query processing. *History*, pages 1–25, 2008.
[46] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.
[47] Chi Zhang, Ryan Marcus, Anat Kleiman, and Olga Papaemmanouil. Buffer pool aware query scheduling via deep reinforcement learning. *arXiv preprint arXiv:2007.10568*, 2020.
[48] Yin Zhang. General robust-optimization formulation for nonlinear programming. *Journal of optimization theory and applications*, 132(1):111–124, 2007.
[49] Yiwen Zhu, Subru Krishnan, Konstantinos Karanasos, Isha Tarte, Conor Power, Abhishek Modi, Manoj Kumar, Deli Zhang, Kartheek Muthyala, Nick Jurgens, et al. Kea: Tuning an exabyte-scale data infrastructure. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2667–2680, 2021.