Intelligent Scaling in Amazon Redshift

Vikram Nathan
vrnathan@amazon.com
Amazon Web Services

Vikramank Singh
vkramas@amazon.com
Amazon Web Services

Zhengchun Liu
zcl@amazon.com
Amazon Web Services

Mohammad Rahman
rerahman@amazon.com
Amazon Web Services

Andreas Kipf*
andreas.kipf@utn.de
Technische Universität Nürnberg

Dominik Horn
domhorn@amazon.com
Amazon Web Services

Davide Pagano
dpagano@amazon.com
Amazon Web Services

Gaurav Saxena
gssaxena@amazon.com
Amazon Web Services

Balakrishnan Narayanaswamy
muralibn@amazon.com
Amazon Web Services

Tim Kraska
timkrask@amazon.com
Amazon Web Services

ABSTRACT

Cloud-based data warehouses are built to be easy to use, requiring minimal intervention from customers as their workloads scale. However, there are still many dimensions of a workload that they do not scale with automatically. For example, in cloud-managed clusters, large ad-hoc queries and ETL workloads must use the same cluster size provisioned for the rest of the workload, and warehouse size does not automatically grow as the underlying data grows in size, causing queries to slow down. In this paper, we describe RAIS, the latest collection of AI-powered scaling and optimization techniques in Amazon Redshift, released in preview at re:Invent 2023, which enable it to scale both vertically and horizontally to adapt to all types of workload variability. RAIS dynamically provisions compute resources to run heavy queries efficiently and automatically optimizes warehouse size for the customer's workload, even as it shifts over time. We show that, depending on the workload, RAIS improves either cost or average query execution time by up to 7.6× and 14.2×, respectively, over existing baselines.

*Work done while at Amazon Web Services.

ACM ISBN 979-8-4007-0422-2/24/06
https://doi.org/10.1145/3626246.3653394

ACM Reference Format:
Vikram Nathan, Vikramank Singh, Zhengchun Liu, Mohammad Rahman, Andreas Kipf, Dominik Horn, Davide Pagano, Gaurav Saxena, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Intelligent Scaling in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3626246.3653394

1 INTRODUCTION

Modern cloud data-warehouses are built for scale and ease of use. They can be set up with a few clicks, they separate compute and storage so that both can be scaled independently, and they save cost by automatically pausing billing when no queries are running. There are two leading approaches to achieve these properties: the first is a managed cluster-based system, such as Amazon Redshift or Snowflake, which uses a set of dedicated compute nodes per customer. These compute nodes are released when idle to save cost, and reacquired when the workload resumes. The second is a shared-compute architecture, such as Amazon Athena or Google BigQuery, in which queries appear to execute on common compute resources and can be individually scaled. These two architectures offer fundamentally different cost and performance characteristics.

The cluster-based model requires a user to select a fixed warehouse size; each size corresponds to a certain hardware configuration with a number of compute nodes. In Redshift Serverless, this size is called the *base capacity*. Selecting the right warehouse is a critical yet non-trivial task for customers; if the endpoint is too small, queries might run slowly; the memory of the cluster might also be insufficient, causing the query to spill to disk and resulting in catastrophic performance. If the warehouse picked is too large, the customer is overpaying for the over-provisioned resources. Even worse, choosing a fixed warehouse size also poses challenges for varying workloads. For example, a customer workload might consist of short running dashboard queries during the day, which can efficiently run on a small cluster, and long running ETL queries during the night, which require significantly more resources. The customer either has to (a) over-provision the cluster to run both types of queries efficiently; (b) manually separate the workload into two data-warehouses, sized for the different workloads, which may be more expensive to maintain; or (c) perform ad-hoc warehouse resizes at the right times.

Most cluster-based data-warehouses offer automatic scaling capabilities to address an increased load on the system, but only in the presence of a large number of concurrent queries. For example, Redshift Serverless offers concurrency scaling, which adds additional compute based on the number of concurrently running queries in increments of the base capacity [12]. Unfortunately, this type of scaling technique does not address all scaling needs. For example, if a large ad-hoc analytical query requires a much larger cluster to execute efficiently (e.g., without spilling to disk), concurrency scaling will not be sufficient. In fact, such large ad-hoc queries can have a severe negative impact on the overall performance of the cluster, since they can occupy compute resources and cause cache thrashing. Additionally, concurrency scaling is not effective when queries become slower as a result of data size increase or other workload changes.

The shared-compute model promises to solve many of these issues by providing the abstraction of allocating resources individually per query. For example, Amazon Athena and Google BigQuery assign compute based on how much data a query has to scan. While the amount of scanned data is not a true measure of the actual work required by a query, e.g., it ignores the cost of joins, it is a proxy.

However, the shared-compute model also has several significant downsides. First, it is not able to cache data as efficiently as the cluster-based model and has more overhead per query [13]. As a result, these systems are often not well suited for short running, dashboard-like queries. To overcome this, some shared-compute data-warehouses use special accelerators to handle these types of workloads [6]. These accelerators can be best described as specialized in-memory database engines or caches, which come with its own set of challenges as they need to be correctly sized and do not support all types of queries. Second, for similar reasons, the shared compute data-warehouse architecture is often significantly more expensive than the cluster-based architecture, especially for active workloads with several concurrent queries [13]. To overcome these limitations, some shared-compute data-warehouses allow customers to reserve capacity. This effectively creates a cluster of a fixed warehouse size, which comes with all the obvious downsides of cluster-based warehouses mentioned above.

In this paper, we describe Redshift's next-generation AI scaling (RAIS) techniques, which were launched in preview at AWS re:Invent'23. RAIS aims to address the shortcomings of the cluster-based scaling model and provide even better scalability than the shared-compute model while maintaining or even improving its biggest benefits, i.e., the leading price-performance and better short-query performance. First, RAIS is able to add compute in different sizes than a fixed warehouse size based on the workload. For example, if a very large ad-hoc query arrives, which the current base capacity is not able to handle, RAIS will automatically allocate a larger amount of resources than the base capacity to efficiently process the query. Second, RAIS observes the workload and automatically determines what the right base capacity should be, which allows for better performance at lower cost out-of-the-box with less experimentation.

However, intelligently scaling up in response to variable workloads is challenging [1]. It requires a model of query execution time on different resource sizes, even if that query has never run on those larger resource sizes before. It requires a design that unifies individual query scaling decisions under a holistic workload-wide optimization, in order to prevent the possibility of cost escalation. Since most improvements in query execution time do not come for free, it requires a way for customers to specify how much cost increase is acceptable for a particular gain in performance. And lastly, it requires understanding the profile of a customer's workload enough to determine when resizing makes sense.

In this paper, we describe how Amazon Redshift Serverless scales both *up* and *out* to intelligently and dynamically size a customer's deployment in the face of all types of workload variability. This feature can be accessed by setting "Price-performance targets" under the Performance and Cost controls section of the Redshift Serverless console.

This paper makes the following contributions:

(1) We describe RAIS, to our knowledge the first production system that is capable of handling *all* types of workload variability with the performance and cost benefits of cluster-based systems. RAIS is a component of Amazon Redshift Serverless.
(2) We demonstrate how RAIS formulates and solves an optimization over cost and performance, by utilizing two components that operate at different timescales: a fast on-demand controller in the query critical path, and a less-frequent global solver.
(3) We highlight the role of three predictive models in RAIS: a resource predictor to estimate memory consumption, a "what-if" scaling predictor that estimates per-query performance on unseen hardware sizes, and a workload

forecaster to help RAIS anticipate predictable query patterns.

(4) We evaluate RAIS on synthetic workloads derived from publicly-available datasets, and show that it offers a favorable tradeoff between cost and performance compared to existing production baselines.

(5) We present a summary of lessons learned and outline key research challenges, some of which we believe are not yet addressed by the research community.

2 OVERVIEW

At its core, RAIS is a serverless offering. Serverless architectures allow customers to automatically scale resource usage to meet their workload demands. Instead of provisioning a fixed set of resources that are always available and billing continuously, a serverless product scales elastically and only bills customers for what they use. RAIS aims to make the correct *scaling decisions* for each query.

2.1 Economics of Scaling

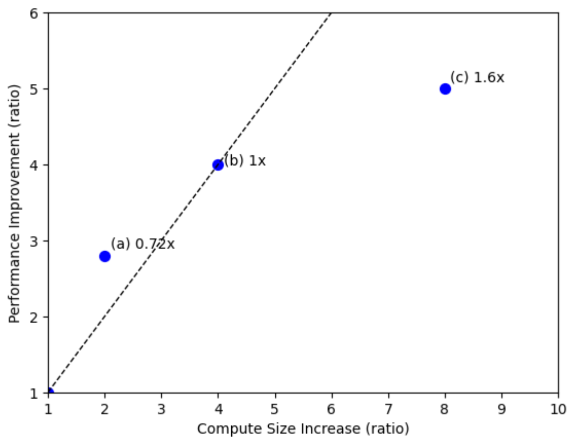


Figure 1: The same query may scale differently at different sizes. The black line indicates linear scalability, and the total cost is marked next to the point. (a) Super-scaling behavior when doubling the compute size, in which case we always prefer running on the larger compute; (b) linear-scaling behavior, so the total cost of the query is identical and we prefer running on the larger compute due to the performance benefits; and (c) sublinear-scaling behavior, where the query's total cost increases 1.6× but it runs $5x$ faster.

When deciding whether to scale a query by using on-demand compute, RAIS must understand the effect of compute capacity on performance. Running a query on a larger cluster can result in better performance, which in turn reduces the time billed. However, the larger cluster also bills at a faster rate since it has more compute or memory. In Amazon Redshift, each cluster bills at a rate specified by its Redshift Processing Units (RPUs), which capture a combination of compute size and memory [2]. These competing effects of cost and performance let us categorize queries into three classes, illustrated in Fig. 1:

(1) *Sub-linear scaling* queries: doubling the RPUs given to this type of query will result in less than a 2× improvement in execution time. Most queries fall into this category, since additional resources have diminishing marginal effects, a property commonly referred to as Amdahl's Law. Running this query on a larger cluster increases cost but also increases performance. Only the customer can decide whether this tradeoff is acceptable to them (see §2.2).

(2) *Super-linear scaling* queries: doubling the RPUs given to this query can result in a performance gain of more than 2×. For example, queries with high memory consumption that saturate their cluster and spill to disk may experience super-linear scaling in regimes where they spill, but sub-linear scaling once the cluster is large enough to avoid spilling entirely (Fig. 1).

(3) *Non-scaling* queries: doubling the RPUs does not decrease query runtime, and may actually increase it. This is commonly observed for short queries with little to no parallelizable work, or queries with substantial network overhead, which increases with cluster size.

It should be noted that all queries will eventually scale sublinearly above a certain level of parallelism. However, it provides a very useful framework to decide if capacity should be increased or not. Unfortunately, the classification of queries as super-scalers, sub-linear scalers, and non-scalers only makes sense when that query runs in a vacuum, ignoring other competing effects apart from resource size. However, this is rarely the case in production workloads. In practice, spinning up additional compute resources for a query has downsides:

- There is a delay between the time extra compute is requested and when it is available to use. As a result, spinning up new compute for short queries can negatively impact their end-to-end latency.
- Queries often benefit from caching effects when re-utilizing the same resources as previous queries. Provisioning new compute for each query reduces the effectiveness of caching.
- Customers can save on cost when multiple queries can be multiplexed onto the same set of resources. Separating queries onto more sets of resources prevents multiplexing and may increase a customer's bill.

Most workloads would achieve better price and faster performance if they had continual access to the same piece of

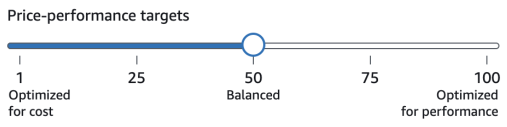


Figure 2: RAIS interface to capture the user's cost-performance tradeoff. Each level of the slider corresponds to a multiple of cost above the minimum cost configuration (level 1) that RAIS may use to optimize query performance.

hardware. We call this continually available set of resources the *base capacity*. The extra compute that RAIS provisions for queries is done on-demand, on top of this pre-allocated base capacity, and only when the base-capacity itself is insufficient to handle the workload. The base capacity may be stopped or paused when it is unlikely to be used.

In theory, super-linear and non-scaling queries are straightforward to provision resources for if RAIS can accurately capture the added latency penalties of spinning up new compute resources. Super-linear queries are both faster *and* cheaper if given more RPUs to run, so scaling up is the obvious choice. By contrast, non-scaling queries are fastest and cheapest on the smallest cluster. We found that sub-linearly scaling queries comprise the vast majority of queries, queries that are faster but more expensive to run with more RPUs. In order to determine whether scaling up is the best decision, RAIS needs to understand how much extra cost the customer is willing to incur for the estimated improvement in performance. This requires customer input, since different customers may value price and performance differently.

2.2 Capturing User Preferences

To capture this tradeoff from the customer - how much is improvement in performance worth? - RAIS includes a slider that the customer can set to one of five different values, from "Low Cost" on the left to "High Performance" on the right (Fig. 2). The farther right the slider is, the more the customer is willing to pay for a fixed improvement in performance. This is the only input the customer needs to provide to RAIS. Note that this type of slider is only one possible way to capture user preferences; another might be to ask the user to set a target execution time, or SLO, for certain types of queries.

Internally, RAIS maps this slider to a single value $\alpha \in (0,\infty)$ that captures the tradeoff between average execution time R and total billed cost C. RAIS then aims to minimize RC^{α}. Intuitively, $\alpha = 0$ corresponds to a configuration that optimizes performance at any cost, while $\alpha = \infty$ prefers being the most cost effective. An intermediate value, like $\alpha = 1$ denotes a proportional tradeoff between price and performance, e.g. a 20% improvement in average runtime should not increase cost by more than 20%. Note that when using product instead of a sum, the scales of R and C relative to each other do not matter, which means that one value α conveniently makes the same kind of trade-offs on different workloads.

Using a customer's historical workload, RAIS simulates its own operation and decisions with various parameters, including base compute capacity. For each set of parameters p, RAIS periodically estimates the corresponding cost C_p and average runtime R_p. For a given slider position, and therefore α, RAIS then chooses the parameters

$$p^* = \operatorname{argmin}_p R_p C_p^{\alpha}$$

to govern its execution.

2.3 Goals and Definitions

Given α, RAIS's optimization goal can be written as:

$$\min_{\{d_j\}} \left(\sum_j R(q_j, d_j) \right) \left(\sum_k r_k t_k \right)^{\alpha} \tag{1}$$

where j indexes over queries q_j in the customers workload, over some time horizon H, and scaling decisions made for each query d_j; k indexes over the resources used, where r_k and t_k refer respectively to the size of the resource in Redshift Processing Units (RPUs) and the duration of time the resource was attached and billable, in seconds. $R(q,d)$ is query response time of query q given scaling decision d, which measures end-to-end latency of the query, including queuing, so it accurately reflects what customers observe.

The queries q_j in the above optimization are entirely customer provided; RAIS can neither choose which queries to execute nor can fully know in advance what those queries will be. While this uncertainty is not fully removable, many production workloads are stable: knowledge of the past can provide an accurate prediction for the future. RAIS uses the predictive power of past workloads to learn the profile of queries that might need to scale in the future.

However, there is always the possibility for surprises: unexpected large, ad-hoc queries or ETL pipelines that ingest an abnormally large volume of data. These are unpredictable, but RAIS needs to be robust to these circumstances to be useful in practice. Therefore, while the solution to Eq. 1 guides RAIS's decisions, the bulk of the decision-making for a query occurs after the query has been issued by the customer, in the query's critical path. Any latency added by RAIS will be observable to the customer, so the scaling decision has a tight latency constraint; RAIS's target is under one millisecond.

Solving a holistic optimization problem like Eq. 1 across an entire workload is infeasible in such a short time, even with full knowledge of the workload. RAIS therefore splits the problem into two parts:

(1) A *Scaling Controller* that runs in the critical path and makes the scaling decision for each query. It is aware only of the query to be executed and key pieces of system state, like the sizes of compute resources attached and the number of running and queued queries. By distilling only the key pieces of information it needs, and exposing a limited set of parameters p that govern its behavior, it can hit its latency target of 1ms. §3 covers the Scaling Controller in more detail.
(2) A *Policy Optimizer* that runs in the background infrequently (1-2 times per day), and is responsible for choosing the parameters p for the scaling controller, and the size of the base compute capacity needed. Since the Policy Optimizer is not latency-bound, it runs a more involved optimization procedure over multiple samples of the forecasted workload over a longer horizon. In order to decide p and the base compute size, the Policy Optimizer has its own model of the Scaling Controller's decision-making process and uses it to simulate what the scaling decisions, and by extension the cost and performance, on a hypothetical workload would look like. §4 describes the Policy Optimizer's operation further.

Although the naming may imply that only the Scaling Controller makes scaling decisions, choosing the correct base compute size is perhaps the most important scaling decision, since it can have an outsize effect on cost and performance compared to the on-demand scaling decisions the Scaling Controller makes for a single query. RAIS must then balance the sizes of the base and on-demand clusters, which have competing effects. If we increase the base capacity, it can handle larger and more-complex queries (e.g., ETL pipelines or data ingestion) on its own; however, it will likely be overprovisioned for light workloads and queries (e.g. dashboard analytics), increasing customer cost during these times of low utilization. On the other hand, a lower base capacity would require us to scale more aggressively on-demand to maintain performance for heavy queries. Trading off between the two requires RAIS to be aware of how a query's cost and performance scales on different compute sizes (§2.1).

2.4 Architecture

Fig. 3 diagrams the interaction between RAIS's components. We describe the high-level organization in this section but leave details to §3 and §4.

The Scaling Controller is part of Reshift's Auto-workload management (Auto-WLM) [12], which allocates resources, such as memory and CPU, to individual queries. The Scaling Controller augments Auto-WLMs capabilities by allowing it to acquire entirely new resources, complete with compute and memory separate from the base compute, to run a query. It is a fast, lightweight decision-making component that lies in the query hotpath, so we require that it adds no more than 1ms of delay to the query's execution time.

The Policy Optimizer is adjacent to but independent of the Scaling Controller, and may decide to change the configuration of the Scaling Controller through the Policy Manager. It may also decide to resize the base compute capacity; this cannot be done immediately since it may interfere with the user's workload. The Policy Optimizer communicates its intent to a Resize Trigger, which watches the user workload to find a suitable time to resize, e.g., when the endpoint is idle. We omit discussion of the Resize Trigger in this paper, for brevity and because it is not a primary contribution of RAIS.

3 ON-DEMAND SCALING

The Scaling Controller is responsible for deciding whether to provision additional resources for a query on-demand, based on the customer's cost-performance tradeoff. The output of the Scaling Controller is a scaling decision d_q for each query q, which comprises the size of the resource to run q on, and whether a new cluster should be acquired if it does not already exist. To reduce resource waste, clusters are detached after a fixed amount of idle time not running any user queries.

The optimization in Eq. 1 cannot be solved simply by considering each query in isolation, since the cost factor is dependent on the choices made on other queries that came before and will come after. For example, if a cluster has already been attached, it may not add any marginal cost to the customer to submit another small query to that cluster. On the other hand, it may be the case that spinning up a new cluster for a particular query is only worth it if there are more queries in the immediate future that could also utilize that cluster. This problem is difficult to solve in the online environment RAIS operates in, particularly at the low latencies required by the query hotpath. We make the problem more tractable by minimizing the per-query *penalty*:

$$d_q = \text{argmin}_d R(q,d,S) C(q,t)^{\alpha} \tag{2}$$

$C(q,t)$ is the cost of all currently active resources, if held for the duration of q's runtime, and S is the current state of the cluster, which includes available memory and number of concurrently running queries.

RAIS cannot perfectly measure all cluster state S nor can it accurately predict how such state affects the execution time of a query; this is a difficult problem (see §5). RAIS simplifies by breaking the response time of a query down into four components.

$$R(q,d_q) = r_{exec} + r_{spill} + r_{queue} + r_{prepare}$$

where r_{exec} is the execution time of query q on the resource indicated by d_q, r_{spill} is an additional additive penalty based on how much this query is estimated to spill to disk, r_{queue} is the estimated time this query will wait in the queue, and $r_{prepare}$

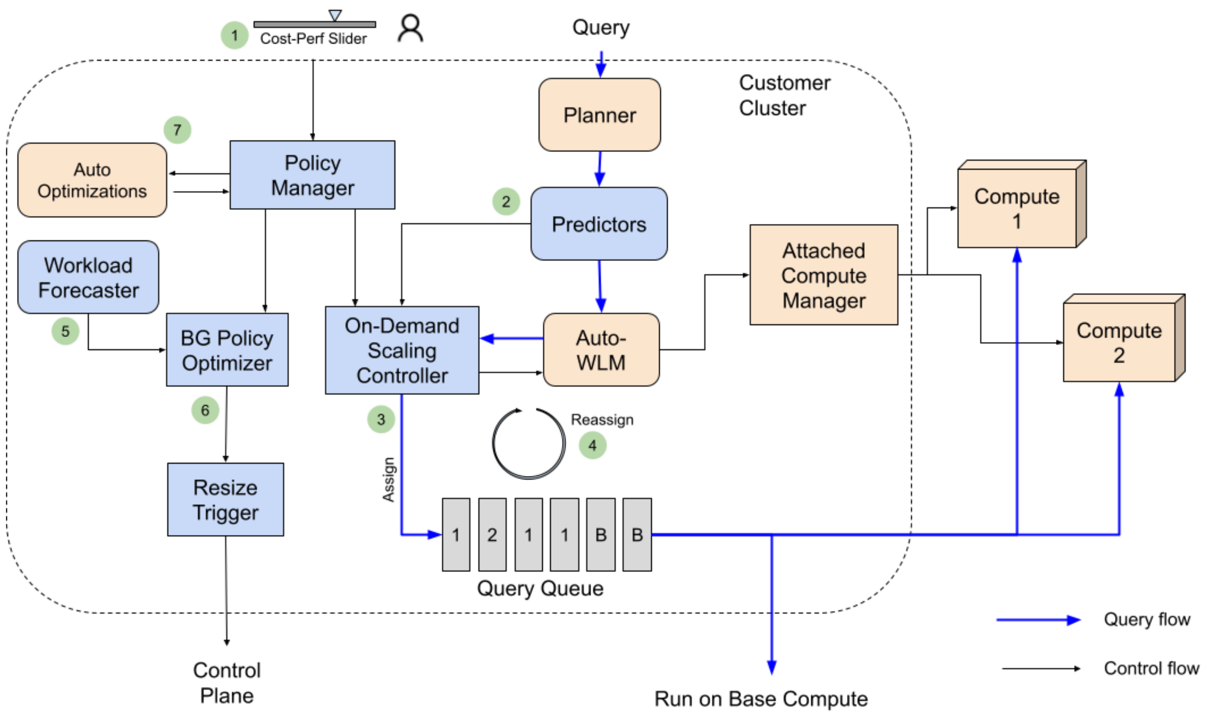


Figure 3: RAIS system diagram. The blue components are those that are new or changed substantially when implementing RAIS. (1) The user specifies their desired cost-performance tradeoff via the Redshift console. (2) Custom predictors estimate a query's memory usage and runtime on various hardware sizes. (3) The scaling controller in the query hotpath decides where to run this query. (4) The query's assignment may change as it queues. (5) The workload forecaster estimates a customer's query load. (6) The forecast is used to optimize RAIS's parameters and possibly resize the base compute capacity. (7) RAIS manages auto-optimizations that may occupy resources but also produce benefit to user queries.

is the time a query is expected to wait for the corresponding resource to be fully prepared and ready to execute queries.

Execution Time. Execution time excludes queuing, delays due to evictions, etc. and is an estimate of the time the query will take to run based on the features derived from the query plan, and the compute size being run on. This is provided directly by the execution time predictor (§3.1). The execution time estimate receives a multiplicative factor to penalize running on a new resource, to reflect cold cache delays.

Spill Penalty. RAIS additionally penalizes queries that require more memory than they can be allocated on the given resource size. We define a parameter γ and for each megabyte over its maximum allowed allocation, RAIS adds penalty of $\gamma e^{-\lambda\alpha} * r$ seconds, where r is the size of the resource in RPUs. This pushes RAIS to be more aggressive to scale if α is low and the query is memory constrained. We tune λ and γ experimentally over the Redshift fleet. RAIS calls this factor out separately from execution time because (a) memory prediction is less affected by transient cluster state, such as number of running queries or cache state, and (b) spilling is a reliable indicator of a super-scaling query. Not scaling a super-scaler is a worst-case scenario for RAIS because the customer sees both slower performance and higher cost; penalizing spilling helps us avoid these situations.

Queuing Delay. Redshift maintains a single physical query queue, but RAIS maintains an abstraction of a separate logical queue per compute resource. High queuing delays may incentivize RAIS to shift a query to a different logical queue, or spin up a new resource entirely if the delay is high enough. RAIS maintains an exponentially-weighted moving average (EWMA) of the *head wait time*, the time between when a query arrives at the head of the queue and when it is dequeued. Simply multiplying this EWMA with the length of the logical queue provides an estimate for queuing delay. Note that Redshift may also scale out if the queuing delay is high; RAIS is aware of this scaling and avoids spinning up too many

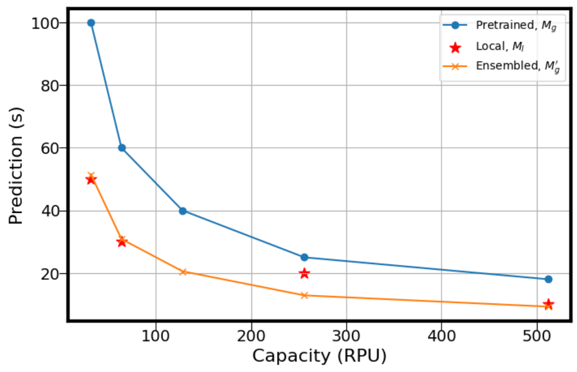


Figure 4: The ensemble predictor (orange) showing the zero-shot trend line (blue) and cache observations (stars). Predictions are normalized.

new resources. However, when α is small, the delay caused by queuing may be enough to spin up a new compute resource. In this situation, RAIS will supersede Redshift's typical scale-out process.

Prepare Delay. Spinning up and connecting extra compute resources takes time before the resource can start running queries. The delay for this is variable and depends on control plane capacity, as well as the database version being used. RAIS keeps a moving average μ and standard deviation σ of prior prepare times observed by the cluster, and treats the prepare time as a normally distributed random variable $T \sim \mathcal{N}(\mu, \sigma)$. If t seconds have already elapsed since RAIS requested a new resource, the remaining prepare delay can be derived:

$$\mathbb{E}[T \mid T > t] = \mu + \frac{2\sigma\phi(z)}{1 - \text{erf}\left(\frac{z}{\sqrt{2}}\right)}$$

where ϕ is the PDF of the standard normal, $z = \frac{t-\mu}{\sigma}$, and erf is the standard error function.

RAIS finds the best scaling decision d_q by enumerating all possible decisions and computing the above penalty. The decisions consist of (a) running the query on the available base compute, (b) running it on any extra attached compute that was previously acquired, (c) queuing it for a resource that was previously requested but not yet acquired, or (d) requesting that a new resource size be acquired for this query. In case (d), RAIS considers a small set of resource sizes: 32, 64, 128, 256, and 512 RPU. This captures the range of scaling effects but is small enough to meet the latency target.

3.1 Execution Time Predictor

This section describes how we make predictions for the execution time of a query on different hardware sizes, while also learning quickly from past mispredictions.

From Redshift's stable production workloads, we found that 80% of queries are repeated. This lightens the burden on the out-of-the-box query prediction model, since we can use recorded result for this query from the previous time it ran. RAIS leverages the query repetition by using a cached prediction engine: queries first check cache for a match, and only if no match is found, the query prediction model is used. Because of this, and because prediction lies in the query hotpath, which is latency sensitive, RAIS uses a lightweight XGBoost model. More details regarding query predictors in Redshift can be found in [3].

However, caching alone is insufficient. The XGBoost model is a "what-if" predictor, estimating the execution time of a query on multiple different compute sizes, most of which the query will not actually run on. As a result, for most compute sizes, the cache is ineffective: simply seeing the query a second time is not enough to provide accurate predictions on all hardware sizes. However, we note that execution time on one hardware size can still provide information about execution times on other hardware sizes. For example, if the XGBoost model predicts a runtime of 60s on 32RPU and 40s on 64RPU, but the cached runtime on 32RPU is 10s, we can reasonably infer that the true runtime on 64RPU will likely be under 10s.

RAIS uses an architecture which ensembles a global "zero-shot" predictor M_g, trained over clusters in the Redshift fleet with diverse workloads, along with a local predictor M_l trained only with data collected from the cluster itself. M_l uses the caching described above, so it provides accurate absolute execution time on hardware sizes it has data for. M_g is trained with more data on more diverse hardware, so we expect it to better capture the *trend* of execution time as a function of size.

We therefore combine M_g and M_l by scaling the trend from M_g to match the absolute execution time observations of M_l, since this uses the strengths of both. Ensembling is a minimization problem to find the weight w that minimizes the loss:

$$L(q, S) = \sum_{s \in S} \left\| w M_g(q, s) - M_l(q, s) \right\|^2. \tag{3}$$

where s are the candidates for compute size. Then the solution for w is:

$$w = \frac{\sum_{s \in S_l} M_l(q, s) M_g(q, s)}{\sum_{s \in S_l} M_g(q, s)^2}. \tag{4}$$

We use $M_g^* = w M_g(q, s)$ as the ensemble model to make a prediction for query q $s \in S$. Fig. 4 compares the shape of the execution time curve as a function of the number of cached data points. Note that the trend, i.e., the shape of the curve, is preserved; it is simply scaled up or down to fit the cached observations.

4 POLICY OPTIMIZER

While the Scaling Controller is responsible for acquiring resources on-demand to run queries as they arrive, the Policy

Scaling Controller inputs	**Forecaster Bucketed Metric**
Arrival time	Number of queries per bucket
Memory usage	Distribution of memory usage for queries in this bucket, as values of P0, P10, . . . , P90, P100 percentiles
Estimated execution time on different compute sizes	For each of the 5 allowed on-demand compute sizes, a distribution of query runtimes for queries in this bucket, similar to above.

Table 1: The metrics required by the Scaling Controller's simulator, and the corresponding metrics collected by the forecaster.

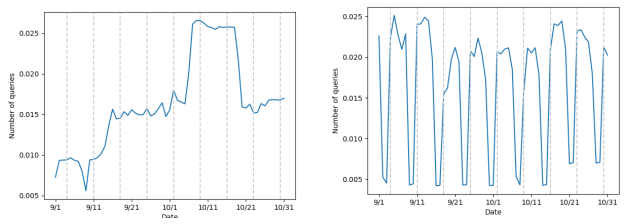


Figure 5: Daily queries (normalized) on real clusters where the "yesterday" model is the most accurate due to long-lasting but non-predictable jumps in load (left), and where the "same-day-last-week" model is the most accurate (right), due to the weekly periodicity.

Optimizer is responsible for decisions on a longer horizon, by default between 12 hours and 1 day. The Policy Optimizer takes as input a customer's historical workload, generates a forecast for the next day, generates multiple query traces from the forecast, and runs a simulation of RAIS's online Scaling Controller to estimate a solution to Eq. 1. The output of the Policy Optimizer is twofold: (a) a target size for the base compute, and (b) a value for α used by the Scaling Controller.

4.1 Workload Forecasting

The goal of workload forecasting in RAIS is to predict enough about the upcoming daily workload so that the Policy Optimizer can make accurate estimates for base compute size and α. The input needed to simulate the Scaling Controller, from §3, comprises only the per-query quantities in column 1 of Tab. 1.

The workload forecaster estimates just these quantities and operates in two logically distinct parts: the data collection phase, and the inference phase. The data collection phase is always active, and sees each query as it arrives. It aggregates the above metrics into 5-minute buckets, as described in Tab. 1. The forecaster persists and backs up this data continuously, in case of restarts or resizes.

At inference time, the workload forecaster keeps several models around. Each model is a particular way of picking a day (or subset of days) within the past week, from which to use the recorded query buckets. The three models we consider are: (1) the previous day, (2) the same day in the previous week, and (3) average statistics from the last week. We chose these models because we observed that most customer endpoints with predictable workloads were either varying with either daily or weekly periodicity (see Fig. 5 for examples). An example of a weekly periodic workload is one that sees spikes on Monday and Wednesday each week, but much lower loads on other days. The model that averages over one week is the best estimator for a truly random workload. More complex models didn't add appreciable accuracy to merit the increase in complexity.

Inference chooses which of the three models to use by evaluating all three for every day in the past week; the model with the lowest mean squared error on the metrics in Tab. 1 is deemed the "winner" and used for the following day's forecast.

4.2 Query Trace Generation

In order to simulate a workload, we need to generate full query traces, by sampling from the distributions present in the forecast, while respecting constraints like the number of arriving queries. RAIS generates multiple traces to simulate different workloads and prevent overfitting to a single arrival pattern.

These traces add noise to the forecast in two ways: first, similar to compression, by aggregating into a forecast, and then upsampling, we necessarily lose some fidelity. Second, we intentionally inject additional noise to the arrival times, memory, and execution time prediction.

4.3 Simulation

The simulator is a model of the system we use for parameter tuning. The two parameters being tuned are the base compute size, which we call M, and the α value used by the Scaling Controller. For the base compute, we use a fixed set of candidates $\mathcal{M}_0$. If the current compute size is m, the complete set of candidates is:

$$\mathcal{M} = \mathcal{M}_0 \cup \{m - 16\text{ RPU}, m, m + 16\text{ RPU}\}$$

The choices for α are always fixed to a candidate set $\mathcal{A}$.

For each parameter combination $(m, \alpha) \in \mathcal{M} \times \mathcal{A}$, and for each trace t, RAIS's lightweight simulation replays t by simulating a base compute of size m, and repeating the Scaling Controller's decision process. The simulation penalizes queries for cold starts (running on a new resource without a hydrated cache), spilling, and if execution happens concurrently with many other queries (due to CPU, memory, and IO contention). The simulation intentionally does not capture low level details, such as IO or CPU contention at the slice or segment level. Its purpose is to account for higher-level interactions and competing effects, such as:

- If a query scales up to a larger resource, how many future queries will make use of that resource? This can improve performance but also increase cost, and varying the α parameter will change this tradeoff.
- If base compute is large, we will likely not need (or want) to spin up extra on-demand compute, pushing the value of α to be higher (less aggressive).
- Underprovisioned base compute should cause spilling and/or excessive queuing, which causes super-linear scaling behavior and bill the cluster more than a larger base compute size.

With average performance $R_{avg}(p)$ and total cost $C(p)$ for each parameter combination p, RAIS chooses p from Eq. 1 as follows: first, we select $C_0 = \min_p C(p)$. Then, if $v \in [0,1]$ is the current performance level, RAIS chooses the p with $C(p) < C_0 \lambda^v$ that has the minimal value of $R_{avg}(p)$. Once p is chosen, RAIS resizes the base capacity to m by finding an appropriate time in the users workload, e.g., when the base compute is idle. It also updates the Scaling Controller to use the new value of α.

5 LESSONS AND TAKEAWAYS

Deploying RAIS involved navigating practical limitations and hurdles of productionizing cloud database systems. Those constraints affect the types of solutions that are feasible and, in turn, affect our approach. We hope the research community will benefit from us sharing them.

Only End-To-End Performance Matters. Improvements to individual components often get lost when combined with the rest of the system. For example, while we could use a more complex and more accurate global query execution time model (e.g, Tree Convolution Neural Network [9]), caching offers a much larger benefit due to the amount of query repetition. Any improvements to the original model would therefore affect much fewer queries and offer a smaller benefit overall.

Never Make the Same Mistake Twice. While it's impossible to guarantee that RAIS will always make the right decision, it *is* achievable to aim to not make the same mistake twice. Scaling the query to a cluster size that didn't actually improve performance will update the value returned by the predictor the next time around. Taking this goal is appealing because it matches what a trained human database administrator can reasonably do.

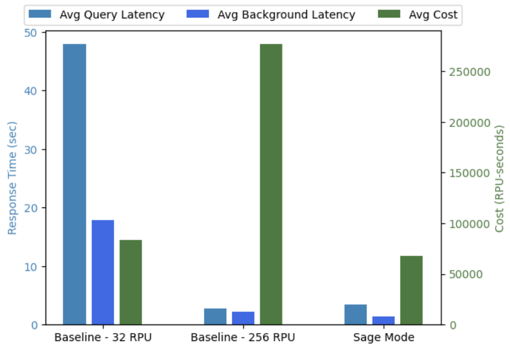
Instance optimization is necessary but not sufficient. Global models and universal algorithms are a long way away from perfect, since we cannot hope to capture the depth of the complexities of a customer's deployment. For example, the pattern of tables available in local cache and the cardinality of predicates that affect join size are only a few of the vast degrees of freedom that are too difficult to account for in RAIS's query prediction and scaling decision. The only sensible solution seems to be defining the parameters that govern the scaling decision, and learn those online from the cluster's operation where possible. At the same time, fully overfitting to the cluster in question ignores learnings from the rest of the fleet, which can only be captured in a global model or decision algorithm. The approach that we found to work best is to pre-train a global solution from the Redshift fleet, but to recognize that each deployment is different and fine-tune that global model locally on the customer's cluster.

5.0.1 How the Research Community Can Help. The research community can help work on difficult challenges that arise in cloud databases and help steer the future of the field. We identified three key research directions that would have direct and actionable impact on cloud databases such as Redshift.

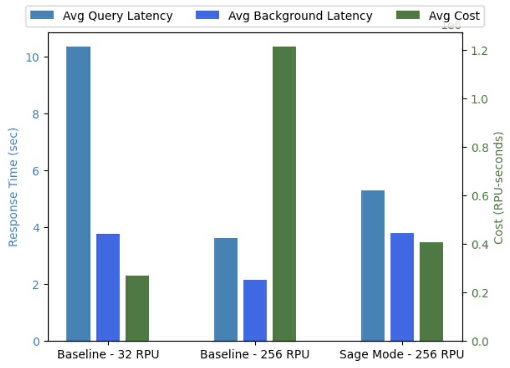
(1) Creating database models for more than just execution time. While we are a long way from accurate execution time prediction models, we are even more lacking in other prediction abilities. For example, memory consumption or CPU requirements based on the query plan would offer a more reliable proxy for performance, since it does not change between runs. Understanding which parts of a query are bottlenecked by the presence of other concurrent queries can improve our understanding of how concurrency affects query performance. Work on "what-if" models, where we probe how a query would perform on a different piece of hardware, is scarce but would be invaluable.
(2) Amdahl's law for the 21st century. Breaking down a query into a serial and parallelizable part is now too simplistic; in a cloud environment, where a query may execute on a combination of hierarchical components, the true scaling behavior is much more complicated. For example, queries can scale superlinearly if they spill to disk, and may not scale at all past a particular compute size. Understanding how queries scale in the cloud can unlock more robust optimization algorithms.
(3) Is the global-plus-local approach to instance optimization the correct one? If it is, which parts of a database system are best tuned via local instance optimization versus via global fleetwide analysis? Or, put differently, which aspects of a database's performance are sensitive enough to data distribution and workload patterns that they merit instance optimizing?

6 EVALUATION

RAIS is implemented in Redshift Serverless as a AI-Enhanced Price-Performance Targets, available in public preview. The settings used in this evaluation are identical to those used in production, with the only exception being fixing a base compute size for particular experiments.



(a) Mixed ad-hoc workload



(b) ETL workload

Figure 6: Cost and performance of RAIS compared to 32RPU and 256RPU baselines on two workloads. RAIS provides either cost or performance benefits compared to the baselines.

We evaluate the on-demand and background scaling components of RAIS against a baseline of Redshift Serverless, which is publicly accessible, with different base compute sizes. We primarily evaluate both systems on performance, considering both average and tail (P95) latency, and billed cost, measured in RPU-seconds. We also report improvements over the baseline in terms of price-performance, defined as the product of average runtime and cost (lower is better). Price-performance is presented for simplicity and captures the particular tradeoff that X% cost increase is acceptable if it results in the same X% performance improvement. Not all customers may share this preference, since it represents only one position on the cost-performance spectrum.

- RAIS scales with query complexity, improving either price or performance of a database deployment by up to 8× and 10.72× respectively, compared to publicly available production baselines. The extent of improvement depends on the properties of the workload, such as the length of time spent running queries.
- RAIS scales with data size by scaling base compute size as it observes longer query execution times. RAIS is able to scale down if overprovisioned or up if underprovisioned. This continual resizing improves price-performance by up to 2×.
- RAIS trades off between cost and performance base on the slider position, so the customer can pay more for faster performance if desired.

6.1 On-Demand Scaling

We ask two questions of the Scaling Controller: (1) can it discriminate between queries of different complexities and (2) does it improve price and performance compared to baselines?

6.1.1 Mixed Workload. We use a synthetic workload with queries from the TPC-DS 3T benchmark. We identify the 20 fastest queries and create a light background stream of 0.2 QPS with these queries. We separately identify a large query (query 78) and issue multiple queries with this template during a 30-minute target window of the workload; these large queries represent ad-hoc queries and have a memory requirement high enough that they spill to disk on 32 RPU. Fig. 6 shows the cost and performances of RAIS restricted to the target window compared with Redshift Serverless at two sizes: 32 RPU, which is underprovisioned for this workload; and 256 RPU, which is overprovisioned. RAIS improves either cost, by 2.94×, or average runtime, by 14.2×. Price-performance improves by between 1.79−9.52×, and we believe this makes RAIS a compelling alternative to existing baselines.

Fig. 7 zooms in on the target window. Static base compute capacities cannot extract the best cost and performance from the database: the 32 RPU cluster handles background queries well by scaling *out* to 3 additional 32 RPU clusters, but the inability to scale *up* hurts performance on two fronts: (1) the latency of the ad-hoc queries suffers without a large enough resource to run on, and (2) the surrounding background workload suffers, since the memory and compute of the cluster are dominated by the large queries. Scaling is able to drastically lower the ad-hoc query latency from 928s to 25s, more than a 37x improvement.

Simply using a larger base compute is also not satisfactory because it heavily overprovisions the cluster, incurring 2.9× the cost of RAIS in this window alone. Since these queries are ad-hoc, they are unpredictable, so customers cannot find the "best" size by trial and error. For cases where mixed workloads are predictable, finding the best size in this way is still costly and cumbersome. RAIS is an easy solution that requires no oversight by the customer.

Note that the length of the workload affects the cost and performance improvement significantly. We conducted a second version of this experiment where the target window is

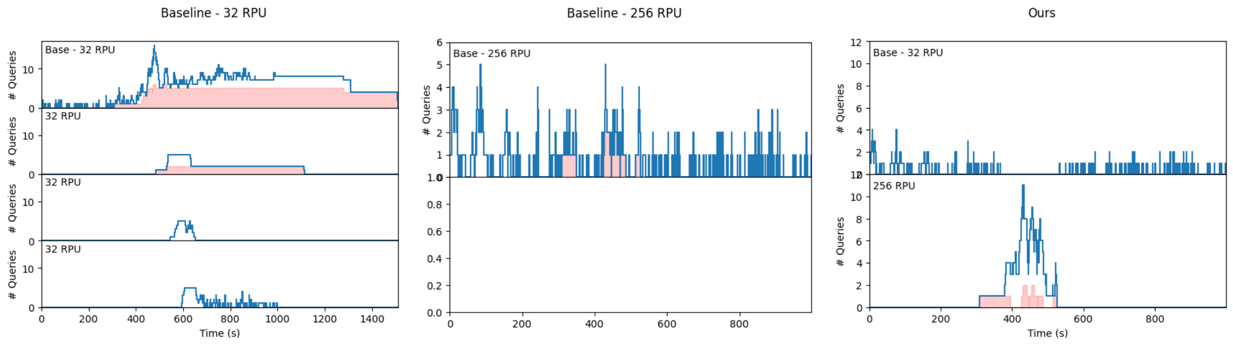


Figure 7: Traces showing the same mixed workload as a function of time, on a 32RPU baseline, 256RPU baseline, and RAIS. The large ad-hoc queries are shaded in red. Note that the 32 RPU baseline takes longer than the other two.

preceded by a long, 12-hour workload of only short queries. We found that the performance improvement over the 32 RPU baseline is smaller (1.48×), but the cost improvement over the 256 RPU baseline is magnified (7.66×). These differences grow further if the workload is extended, e.g., to a full 24 hours.

6.1.2 ETL Workload. Another important case for RAIS is daily ETL jobs, which are often large and too heavy for a cluster that is solely provisioned for, e.g., shorter dashboarding queries. We run an ETL-style workload, with 6 large copy queries arriving at the end of a light workload consisting of short queries. Fig. 6 shows the price and performance tradeoffs that RAIS makes compared to the same baselines. However, this case is different from the ad-hoc workload in §6.1.2 since the large copy queries do not overlap with the short queries, removing the need for performance isolation. We see between $2-3\times$ improvement in either cost or performance, and like before, the scale of these improvements depends heavily on the length of the short query workload.

6.2 Resizing Base Compute

To evaluate the ability of RAIS to scale base compute, we run workloads that benefit from resizing. Decisions to resize are controlled by the Policy Optimizer. We test resizing in two directions: (a) does RAIS scale the base compute down when the workload is too light? and (b) does RAIS scale the base compute up as the table size increases and makes queries slower?

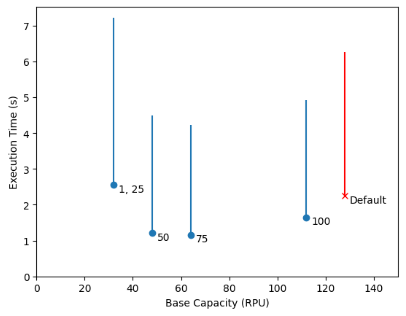
Scaling Down. Starting with a default endpoint size of 128 RPU (the Redshift Serverless default) and with a performance level of 1 (most cost conscious), we run a TPC-DS 100G workload on repeat, every 10 minutes. After running for 12 hours, RAIS evaluates the base compute capacity and decides to resize. Fig. 8(a) shows the cost and performance of the workload on 128 RPU (in red) and the actual cost and runtime for the 5 different slider positions in blue (note that positions 1 and 25 map to the same base compute size in this case). Importantly, the size of the base compute decreases as the slider level goes down, and performance improves as the slider level goes up. The exception is at slider level 100, corresponding to a base compute of 112 RPU, at which point performance starts to degrade since the cluster is too large for the workload. RAIS recognizes that performance is better at smaller cluster sizes for this workload and does not select anything larger.

Scaling Up. We reverse the process by starting with a small base compute of 32 RPU. We run a light 15-minute workload based off of TPC-DS 3T, consisting of queries that each execute in under 2 seconds, in isolation. Each day, we insert a copy of the TPC-DS 1T fact tables (those that end with "-sales" or "-returns"), and rerun the workload. We set the performance slider to 1, indicating a cost-conscious customer. Fig. 8(b) shows that as we increase the table size each day, the query execution times increase, since each query must now scan more data. RAIS searches for the minimum-cost configuration each day but makes no changes on the first two days. On day 3, RAIS deems that the original size of 32 RPU is too small, and dynamically sizes the endpoint up to 64RPU. As a result, the runtimes drop by a factor of more than 2×.

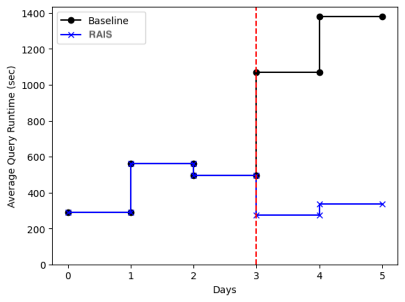
The performance improvement in Fig. 8(left) is not sufficient to show that RAIS is resizing correctly. For example, RAIS could choose a large compute capacity, such as 256 RPU, which might speed up queries but charge too much for this cost-conscious customer. RAIS must choose to resize only when it is appropriate, and to a size that will not escalate cost unnecessarily. Fig. 8(c) shows that the workload's price-performance improves by 2× after RAIS resizes, indicating that the decision to resize was preferable to the baseline.

6.3 Changing Performance Level

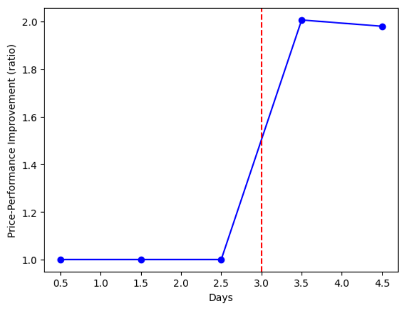
When a customer changes their cost-performance tradeoff via the slider, they implicitly change the value of α used by the Scaling Controller: a higher value of the slider results in



(a) Resizing down

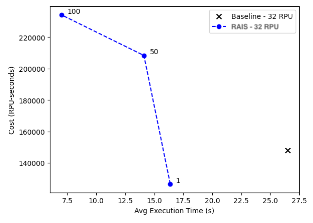


(b) Runtime as data grows



(c) Price Performance Improvement

Figure 8: (a) RAIS scales down to a smaller base compute if overprovisioned (red), based on slider level. Resize candidates' average runtimes are shown in blue, with bars up to P95, and are annotated with the corresponding slider level. (b) On a consistent workload of short TPC-DS queries as the data size grows each day, performance of queries degrades as the data size increases but stabilizes after RAIS resizes. (c) Resizing improves price-performance.



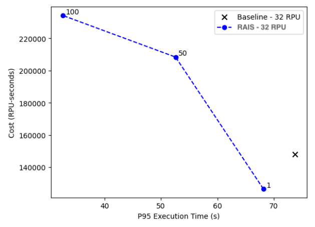


Figure 9: Cost and performance (average - left, P95 - right) of TPC-DS 3T on a 32 RPU base capacity, annotated their respective slider levels. $\alpha = 2.25$ **for slider position 1,** 1.25 **for 50, and** 0.25 **for 100.**

a lower value of α, causing RAIS to acquire extra compute resources more aggressively. Here, we evaluate whether that change in α has the desired effect: a higher α should result in faster performance but at a higher cost.

Fig. 9 compares the performance of the Scaling Controller against baseline runs of Redshift Serverless on a TPC-DS 3T dataset, using all queries from the standard TPC-DS workload in a closed-loop pattern, i.e., the next query is issued when the previous one finishes. Since the is not used in this experiment, we fix the base capacity to 32 RPU for both RAIS and the baseline. RAIS achieves either lower cost or better performance than the baseline configuration, and sometimes both, corroborating the price-performance results from §6.1.2. In particular, using average runtime as our performance metric, RAIS achieves price-performance improvements of 1.89×, 1.34×, and 2.4× at slider levels 1, 50, and 100, respectively, compared to the baseline. Fig. 9(b) shows similar improvements with P95 runtime. Importantly, from the point of view of the customer, as the slider level increases, i.e. as α decreases, RAIS's performance improves at the expense of higher cost.

7 RELATED WORK

Workload Management. Prior work on workload management and scheduling has considered optimizing user-set performance objectives [4, 5, 10], some using reinforcement learning [8, 11]. To the best of our knowledge, none are production systems that address practical difficulties of cloud deployments; solutions either work on a single cluster, rely on known query templates, or need accurate execution time and resource consumption predictions. RAIS builds on traditional workload management, including Auto-WLM in Amazon Redshift [12]. In particular, it goes beyond CPU and memory provisioning to acquire and release resources at the scale of an entire cluster.

Parameter Optimization. RAIS tunes base warehouse capacity and scaling aggressiveness in an offline manner similar to auto-tuning database systems [7, 14]. However, there are two key differences. First, in a production setting, RAIS cannot run experiments on customer clusters to learn the best configuration, which makes controlled exploration difficult; and second, base capacity cannot be changed easily without disruptions to the customer. As a result, RAIS must rely on simulations to make these configuration choices.

8 CONCLUSION

We explained the techniques behind Redshift's AI-driven scaling and optimizations (RAIS), a feature of Amazon Redshift Serverless that provides dynamic vertical scaling and automatic base compute resizing. RAIS can intelligently adapt to different types of workload variability, expanding the capabilities of modern cloud data warehouses. We showed that, compared to existing baselines, RAIS can offer customers significant improvements in price and/or performance without requiring any manual intervention or workload partitioning.

REFERENCES

[1] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J Elmore. 2011. Database scalability, elasticity, and autonomy in the cloud. In *International conference on database systems for advanced applications*. Springer, 2–15.

[2] AWS Amazon. 2023. *Amazon Redshift pricing*. https://aws.amazon.com/redshift/pricing/.

[3] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[4] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüş. 2011. ICBS: Incremental Cost-Based Scheduling under Piecewise Linear SLAs. *Proc. VLDB Endow.* 4, 9, 563–574. https://doi.org/10.14778/2002938.2002942

[5] Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Junichi Tatemura. 2011. SLA-Tree: A Framework for Efficiently Supporting SLA-Based Decisions in Cloud Computing. In *Proceedings of the 14th International Conference on Extending Database Technology* (Uppsala, Sweden) *(EDBT/ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/1951365.1951383

[6] Google Cloud. 2023. *What is BI Engine?* https://cloud.google.com/bigquery/docs/bi-engine-intro.

[7] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with ITuned. *Proc. VLDB Endow.* 2, 1, 1246–1257. https://doi.org/10.14778/1687627.1687767

[8] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[9] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*. 1275–1288.

[10] Ryan Marcus and Olga Papaemmanouil. 2016. WiSeDB: A Learning-Based Workload Management Advisor for Cloud Databases. *Proc. VLDB Endow.* 9, 10, 780–791. https://doi.org/10.14778/2977797.2977804

[11] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1228–1242. https://doi.org/10.1145/3514221.3526158

[12] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan Narayanaswamy. 2023. Auto-WLM: Machine learning enhanced workload management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data*. 225–237.

[13] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing a Cloud DBMS: Architectures and Tradeoffs. *Proc. VLDB Endow.* 12, 12, 2170–2182. https://doi.org/10.14778/3352063.3352133

[14] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029