Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools

Matthew Perron
MIT CSAIL
USA
mperron@csail.mit.edu

Raul Castro Fernandez
University of Chicago
USA
raulcf@uchicago.edu

David DeWitt
MIT CSAIL
USA
david.dewitt@outlook.com

Michael Cafarella
MIT CSAIL
USA
michjc@csail.mit.edu

Samuel Madden
MIT CSAIL
USA
madden@csail.mit.edu

ABSTRACT

Analytical query workloads are prone to rapid fluctuations in resource demands. These rapid, hard to predict resource demand changes make provisioning a challenge. Users must either over provision at excessive cost or suffer poor query latency when demand spikes. Prior work shows the viability of using cloud functions to match the supply of compute to the workload demand without provisioning resources ahead of time. For low query volumes, this approach is less costly at reasonable performance compared to provisioned systems, but as query volumes increase the cost overhead of cloud functions outweighs the benefit gained by rapid elasticity. In this work, we propose a novel strategy combining rapidly scalable but expensive resources with slow to start but inexpensive virtual machines to gain the benefit of elasticity without losing out on the cost savings of provisioned resources. We demonstrate a technique that minimizes cost over a wide range of workloads, environmental conditions, and compute costs while providing stable query performance. We implement these ideas in Cackle and demonstrate that it achieves similar performance and cost per query across a wide range of workloads, avoiding the cost and performance cliffs of alternative approaches.

ACM Reference Format:
Matthew Perron, Raul Castro Fernandez, David DeWitt, Michael Cafarella, and Samuel Madden. 2024. Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Analytical query workloads can have rapid fluctuations in resource demand. This can be caused by a set of reporting queries submitted simultaneously, a join output requiring a large amount of memory, or a new ad-hoc query operating over a massive dataset. This is especially challenging in data lakes where input data can be read from any source in cloud storage at any time, making query input sizes, and associated resource demands, unpredictable. Cloud function services such as Amazon Lambda [10] allow users to rapidly marshal thousands of cores of compute and terabytes of memory in a matter of seconds. Some prior work [24, 27] has demonstrated the low cost and reasonable performance of this approach for workloads with low query volumes, even when intermediate state needs to be exchanged through cloud object storage.

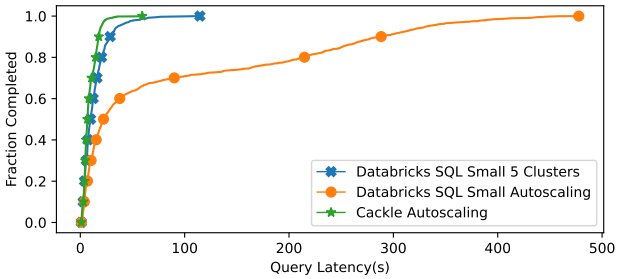

Figure 1: Latency of queries on Cackle, Databricks SQL `small` with five clusters, and Databricks SQL Auto-scaling in an hour-long workload of 1500 TPC-H queries.

As query volumes increase, the cost savings due to elasticity are eclipsed by the cost premium of using elastic resources. Because a large query may arrive at any moment, resource demands fluctuate in rapid, often unpredictable, bursts. These bursts remain even in workloads with many concurrent queries. Workloads that contain primarily user generated queries are particularly difficult to predict as queries may arrive at irregular intervals.

While current popular analytical query processing systems have auto-scaling features to handle changes in demand, this happens at a coarse granularity, adding a few nodes or a cluster at a time, only *after* queries are queued, and removing these resources after idle for several minutes. As new nodes and clusters take tens of seconds to minutes to come online, and only are spawned after a cluster is saturated [5, 23, 26, 29], these auto-scalers are poor at responding to transient workload changes, and queries suffer significantly diminished performance until new hardware can be provisioned. Furthermore, clusters or nodes that are spun up to match an increased demand are slow to shut down [1], presumably to avoid frequently starting up and shutting down nodes. For example, in a Databricks SQL Pro small warehouse with auto-scaling,

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the 80th percentile of queries is more than an order of magnitude slower than on an over-provisioned warehouse with five clusters. We show a CDF of the queries in this workload in Figure 1.

In this paper, we instead propose a novel hybrid technique, called Cackle, where persistent demand is serviced by provisioned virtual machines, but rapid demand spikes are serviced with elastic pools of resources, such as AWS Lambda [10]. The elastic pool allows a system to be highly responsive to demand without incurring a significant increase in cost. Thus, we avoid the cost instability of fixed over-provisioning and the performance instability of current auto-scaling approaches. However, this comes with several challenges. In particular the system must handle rapid changes in resources. Cackle must make allocation decisions to minimize cost under a range of workloads, cost conditions, and environmental conditions. We use a strategy that adapts to these varying conditions by choosing among a family of simple strategies over the measured historical workload. We show that our approach gains the benefit of rapid elasticity of resources without losing the performance benefit of over-provisioning, and adapts well to a variety of workload and environmental situations. For example, Cackle achieves similar performance to an over-provisioned Databricks SQL warehouse despite starting with no compute running at the start of the workload, as we see in Figure 1. We will also show that Cackle achieves cost stability across a range of workloads.

In summary, the contributions of this work are (1) a demonstration of the cost and performance stability of this technique under a range of different workloads and environment variables; (2) an analysis of the impact of varying cost characteristics of the components of this model; (3) a novel approach for minimizing cost using a family of simple strategies; (4) an implementation of a hybrid query execution engine that validates the findings of our model and technique; (5) experimental results showing that elastic pools allow systems to achieve cost and performance stability across a range of workloads in a real implementation.

2 BACKGROUND AND MOTIVATION

Analytical query workloads are comprised of a spectrum of queries of differing cost, and with differing latency tolerances. Some batch queries can have their execution delayed for hours to save on cost, while many ad-hoc queries would impair analyst productivity if delayed too long. Understanding this, modern cloud analytical DBMSs allow users to place queries into queues of different priorities. This can be effective at allocating available resources preferentially to critical queries. But if there are a sudden influx of latency-sensitive queries and not enough resources to serve them, then queries inevitably experience higher delays. Because these queries are submitted at irregular intervals it is difficult to predict when these spikes in demand will occur.

This query mix results in a resource demand curve over time. A provisioning of resources that falls below this curve results in delayed queries, missed SLAs and overall poor user experience. On the other hand, a provisioning that exceeds demand results in excessive cost. Because of the inherently difficult to predict nature of query arrivals and when demand for resources will spike, choosing a good provisioning is a challenging task.

While modern cloud analytical systems have auto-scaling features that can respond when a system is overloaded, allocating new clusters or nodes takes tens of seconds to minutes. To avoid these slowdowns in important queries, system administrators can over-provision, but this means paying a consistently higher cost for these systems just to handle demand spikes for that may only last a few seconds.

With an infinitely scalable pool of compute resources that can instantaneously start and stop, systems can service this underlying demand curve exactly. Queries can execute immediately, never sitting in a queue, and idle resources can be immediately stopped, avoiding the cost of over-provisioning. Such pools of resources are already available. Cloud function services like AWS Lambda can allocate new resources extremely quickly, and prior work [24, 27] has shown how they can be used to build query execution engines.

Starling [27] and Lambada [24], showed that the high per-query costs of these rapidly scalable systems made them too expensive for all but the most infrequent workloads. This is the result of two factors. First, the cost of a time-unit of compute on cloud functions ranges between 3 and 10x as much as the equivalent on-demand or spot priced instances. Second, because of networking limitations on the elastic pool, data must be exchanged through an intermediate service, such as Amazon S3 [6], driving up cost. An ideal system would have the elasticity of an unlimited pool of compute with the cost of a provisioned system.

The purpose of our technique is to deliver on this goal, to minimize the cost of a system for serving a wide range of workloads and environmental conditions, such as varying startup times of new virtual machines and the relative costs of elastic pools versus provisioned servers. Furthermore, the system should be robust when the workload or environmental variables change, and should execute all queries with performance similar to a dedicated, right-sized cluster.

2.1 Workload Examples

OLAP workloads tend to process vast volumes of data. Running a single new query on a large volume of data may significantly increase the resource demand of the system. Modern data architectures make this problem more difficult as data is stored long term in cloud storage services like Amazon S3 [6], Google Cloud Storage [18], or Azure Blob Storage [11]. Therefore, a query can be executed against any data in cloud storage at any time.

The predictability and performance demands of queries on OLAP systems vary widely. We classify queries in three broad categories.

Batch Queries These include queries that can tolerate high latency and can thus be executed on whatever spare, inexpensive resources are available. For this type of query, delaying queries while resources start up is not an issue. These queries might include regular reports or even system maintenance jobs. These queries will benefit least from the availability of an elastic pool of compute, as the lack of tight latency requirements means that it is less expensive to wait until virtual machines can be started.

Periodic Queries These include queries from a template that are submitted at regular intervals, often originating from dashboards or other automated sources. Queries of this type are highly predictable in their arrival time and resource requirements. These queries may gain some benefit from elastic pools depending on

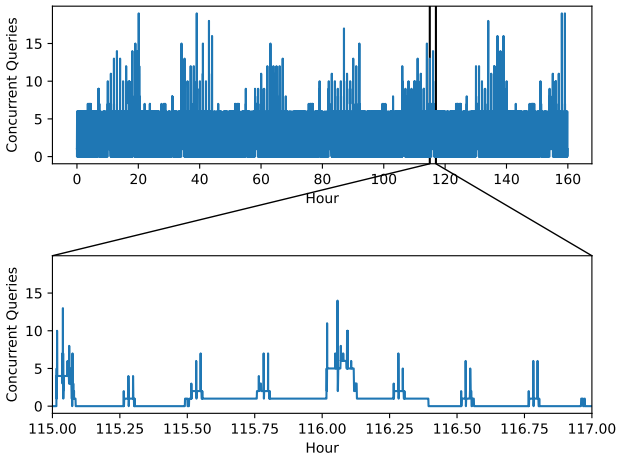

Figure 2: Number of *concurrent queries* in the startup workload with the full workload above and a two-hour window of the workload below.

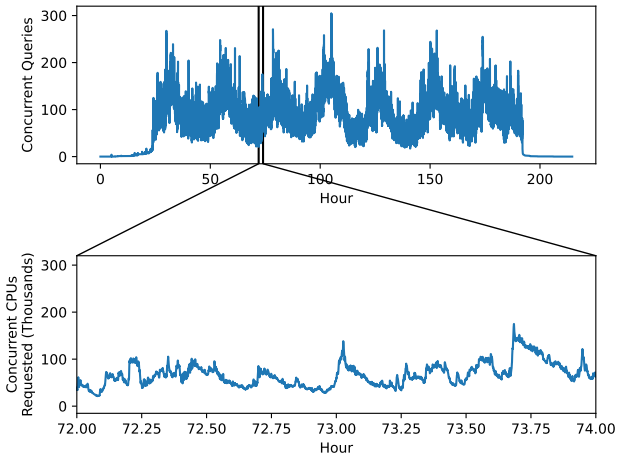

Figure 3: Number of *concurrent CPUs* requested in the Alibaba 2018 workload with the full workload above and a two-hour window of the workload below.

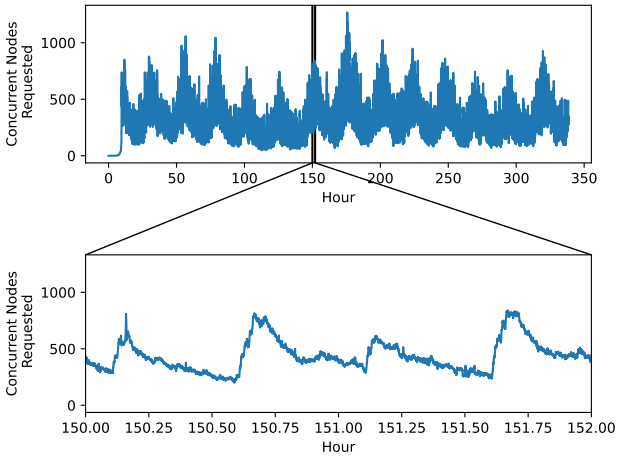

Figure 4: Number of *nodes* requested in the Azure Synapse 2023 workload with the full workload above and a two-hour window of the workload below.

their resource requirements and the frequency of queries. The less frequent the queries, the more helpful an elastic pool will be.

Interactive Queries These queries include those submitted by a user directly or through a set of tools. An example of this might be a data scientist using Tableau [30] to analyze a dataset. These queries are characterized by larger variability in arrival times, data set sizes, and query complexity. User productivity depends on these queries completing within a few seconds. These queries would expect to see the largest benefit from an elastic pool of compute. Since they are human-generated, arrival times, rates, and input sizes of these queries are difficult to predict. While there may be repeated executions of some queries within an interactive session as users refine or drill down on previous results, the arrival of sessions themselves is quite unpredictable.

Below we present traces from three real-world workloads that demonstrate a mixture of the above query types. These traces do not include individual queries, but instead contain data on their resource utilization or query start and end times. While not a perfect sampling of analytical query workloads, these workloads highlight key properties of analytical query workloads:

(1) Workloads contain rapid spikes and drops in demand for resources that are difficult to predict.
(2) Workloads contain periodic increases and decreases in demand in cyclical patterns.
(3) Demand spikes and periodic workload changes can cause queries to queue until resources become available.

Unfortunately the metrics provided by the owners of these systems differ and cannot be compared directly without major assumptions. However, each provides evidence of periodicity and the unpredictability of resource requirements and query arrival rates.

2.1.1 Startup Workload. The workload in Figure 2 is a trace of queries from an AWS Redshift data warehouse at a startup over the course of a week in May 2022. The trace contains only the start and end time of queries. The queries running against this database are a mixture of analyst queries and dashboards. Thus, there is some regularity. Interestingly this workload leaves the cluster idle or with a single query most of the time and the cluster is over-provisioned in order to meet performance requirements. The figure shows a plot of the number of concurrent queries executing on the warehouse.

2.1.2 Alibaba 2018 Workload. The 2018 Alibaba cluster trace [3] provides data on jobs and tasks in a cluster computing workload over the course of a week. We measure the number of concurrent CPUs requested by all jobs in the system. This allows us to see the daily periodicity of the workload. Like the startup workload above, there is a periodic increase in resource demand each day, with unpredictable large spikes in demand that happen regularly.

2.1.3 Azure Synapse SQL Workload. Figure 4 shows the trace of a workload on an Azure Synapse SQL Cluster [2, 12] over the course

of two weeks in 2023. The plot shows the number of concurrent nodes requested for all queries submitted to the system. In this data warehousing workload, we see resource demand peak each day, with slightly more demand on weekdays than weekends. In addition, this workload has rapid unexpected spikes at irregular intervals that cause resource demands to double or triple in the course of a few minutes. These rapid changes in workload lead to many queries queuing before execution.

2.2 Elastic Pools

Given the rapid changes we see in real world workloads, an elastic pool of resources can help to alleviate poor performance caused by rapid demand spikes. An elastic pool of compute resources should grant users near instantaneous access to as many resources as requested, allowing a user to burst out computation. In general, we expect elastic pool resources to be available within a second. General purpose elastic pools are available in cloud function services like Amazon Lambda [10] and Google Cloud Functions [17].

Elastic pools can also be built on a service level. For instance, Redshift Spectrum [13] is a system in production today that makes use of a large multi-tenant pool of workers shared between users in a cloud region and is used to read user data from external tables.

Our technique requires elastic pools to have two properties:

(1) Immediate Availability: Resources must usable at a latency low enough that the performance of the workload is minimally impacted. This ensures that the resources are available as soon as demand rises.
(2) Fine-Grained usage: Resources should not be billed after they become idle. This keeps the cost of using the pool low.

Existing services like AWS Lambda provide these properties, but at an additional cost that covers the operation of the resource when it is not provisioned to a user. As a result, resources from an elastic pool are more expensive per unit than their provisioned equivalents. Running a function on AWS Lambda with one vCPU and 4GB of DRAM costs $0.24 per hour [10] while the spot price of a similar VM, `c5.large` in `us-east-1` was only $0.04 in Feb 2023 [4], six times less expensive. However, these prices are subject to change over time, for example, in April 2023, the price of the same instance increased to $0.06, an increase of 50%, while an on-demand instance was $0.085. Any system that uses these elastic pools must be sensitive to these cost fluctuations.

Because of the cost premium of using an elastic pool, it is inefficient to exclusively use these resources, particularly when the cost differential is high. Elastic pools are best at servicing demand that is unexpected and cannot be delayed easily. As we will show in Section 5, using elastic pools to handle bursts, rather than over-provisioning dedicated instances, can be less expensive for many workloads despite their higher cost per unit of resources.

3 HYBRID SYSTEM DESIGN AND ASSUMPTIONS

Cackle is designed to address the resource spikes that are endemic to analytical query workloads while minimizing the overall cost of the system. It achieves this by using both provisioned resources and elastic pools. It chooses a provisioning of resources that minimizes cost for a wide range of workloads and environmental conditions. In this section we describe the system design of Cackle and the assumptions that inform this design. Cackle is a query execution engine. It receives physical query plans and executes the plan on available resources. As batch jobs can be delayed until provisioned resource become available, we focus on reporting queries and ad-hoc user-generated queries where low query latency is essential.

Cackle executes queries by breaking up its operators into a DAG of stages. Each part of a query executes as a *stage* with one or more *tasks* that can be executed concurrently. Task sizes are chosen so that they fit into fixed sized containers. Data must be exchanged between stages. Each task processes a subset of the stage's input data and produces an output to be consumed by future stages or the end user. Tasks are scheduled on available operators, and run to completion. Each stage must wait for all tasks in its upstream stages to complete before its tasks are eligible to be scheduled. This execution model is similar to Spark or Presto.

Cackle has three components responsible for coordination, execution, and shuffling respectively. We explain each layer's responsibilities and interactions below.

Coordinator. The coordinator has two main responsibilities. First, it receives query requests, and schedules computation on provisioned machines and the elastic pool depending on the resource availability. Second, it monitors the workload and makes resource provisioning decisions to minimize the overall cost of the workload during execution. It runs on a single small provisioned VM.

Execution Layer. The execution layer is composed of provisioned VMs and an elastic pool of compute resources. Tasks are scheduled first on available provisioned VMs. If none are available then the task is executed on the elastic pool. We assume that provisioned machines have a delay between their requested time and the time they are provisioned and able to execute tasks. Once started, we are billed for their usage until terminated. VMs have a minimum billing time of one minute, meaning there is no value to shutting started idle instances down unless they have been running for at least that duration. Compute from the elastic pool is available instantly and is billed at the millisecond granularity. We assume that the cost per second of the elastic pool is some factor more expensive than the equivalent time on a VM.

Shuffling Layer. The shuffling layer uses provisioned virtual machines dedicated to temporarily storing intermediate shuffle state. The shuffling layer is required in Cackle as the elastic pool of compute we use, AWS Lambda, is unable to exchange data directly. It also allows the system to independently scale intermediate storage and compute. We call these provisioned virtual machines in the shuffling layer "shuffling nodes". When insufficient shuffling nodes are available, the compute layer of Cackle instead shuffles through cloud object storage systems like Amazon S3 using the method outlined in Starling [27]. We use a mix of VMs and S3 to reduce the cost of shuffling. VMs are used to reduce the cost of S3 requests and S3 is used to absorb excessive requests. Shuffling nodes are billed in the same way as provisioned virtual machines in the execution layer. The elastic pool of the shuffling layer uses cloud object storage and is billed for each read and write request.

We describe how queries are executed using the above layers below and the necessary assumptions of Cackle below.

3.1 Work Scheduling and Query Execution

Query execution begins when a user submits a query execution plan to the coordinator. The plan consists of a directed acyclic graph (DAG) of stages. Each stage has one or more tasks that must be executed to completion before its dependent stages can be scheduled. The coordinator begins by scheduling tasks in stages without dependencies, e.g., base table scans. When a stage's dependencies have completed, its tasks are scheduled. This is repeated until all stages have completed and results are returned to the user.

Unlike other query execution engines that use this DAG of stages execution model, such as Apache Spark [32], tasks never wait in a scheduling queue. This is possible through the use of the elastic compute pool. This execution model assumes that requests made to the elastic pool are satisfied quickly enough that it has minimal impact on a query's overall performance. We also assume that a given task will have an equivalent runtime on similarly sized compute from the elastic pool and on a provisioned instance.

Like other cloud analytical query processing engines, base table data is stored and read from cloud object storage services. Intermediate data, however, is either exchanged through the shuffling layer. The coordinator chooses a provisioning of shuffling nodes by observing a trace of the volume of data shuffled and the number of writes to cloud storage.

3.2 Cost Models and Environmental Conditions

The coordinator is responsible for responding to varying workload, cost, and environmental conditions. While the system assumes that performance will be equivalent regardless of execution environment, the cost can be vastly different. If the demand for compute is flat over time, provisioning virtual machines to service that demand will be less expensive than using compute from the elastic pool. It is the responsibility of the coordinator to minimize the cost of query execution given the workload and execution environment.

The cost of a given workload and resource provisioning are subject to the cost model and other properties of provisioned resources and the elastic pool. In our work we assume the cost model of Amazon Web Services including Lambda [10], EC2 Spot Instances [4], and S3 [6]. For instance, AWS Lambda cloud functions typically begin within 100ms and are billed at the millisecond granularity. However, if the properties or cost model of the underlying services change then a system has to adapt to these changes. The techniques we describe in the following sections can be adapted to new scenarios as long as there is an elastic pool of resources that are granted to the user instantly, the cost models of both provisioned resources and the elastic pool are known, and the behavior, in particular the time to start new provisioned resources is predictable. Given these assumptions we describe considerations for a strategy to minimize cost in the following section.

4 APPROACHES TO PROVISIONING

Most cloud analytics systems delay work until provisioned resources become available, but because Cackle is targeted at interactive workloads, it executes work as soon as it arrives. Therefore, **the goal of an allocation strategy for Cackle is not to improve query latency or performance but to reduce overall cost**. In this section we describe the main factors that impact cost, and develop a strategy able to respond to the properties of analytical workloads we described in Section 2. We begin by describing the cost and environmental factors that impact strategy design.

4.1 Cost Factors

We assume that any given task executing in the compute layer has access to the same compute resources and its performance is the same amount regardless if it is scheduled on a provisioned virtual machine (VM) or in the elastic pool. We assume that the cost structure is the same as that on AWS [7]. Specifically, provisioned VMs have a latency between the time they are requested and the time they become available to execute tasks. Users are charged for these instances from the time they startup until the time they are shutdown, but have a minimum billing time. In AWS we have observed that the time to start tens of VMs is around three minutes. The minimum billing time is set to be one minute. The elastic pool is available in less than a second, is charged at a millisecond granularity, and has no minimum billing time. However, as we describe above, this flexibility comes with an increased cost.

Thus, a strategy should choose, moment-by-moment, a *target* allocation of VMs. If the target is less than the number currently allocated, nodes will be terminated once idle. If the target is larger, then VMs will be requested from the cloud provider, and will start after the delay time. If the resource demand is larger than the current allocation, work will be scheduled on the elastic pool.

We consider two naive strategies and explain why they are a poor match for these conditions and present our approach below.

4.2 Fixed Strategies

A fixed strategy chooses a provisioning of virtual machines at the beginning of the workload and the target remains fixed throughout workload execution. Thus, the number of virtual machines never changes. A challenge with fixed strategies is choosing the correct fixed provisioning, choosing too large or too small a provisioning can have a large impact on cost. Obviously, a fixed strategy does not change with the workload and, in a periodic workload, may be significantly under-provisioned when the workload peaks, and over-provisioned during workload troughs. Ideally a strategy should adapt to the workload as it changes, and should not require a user to know the details of their workload a priori.

4.3 Workload-Adaptive Strategies

Instead of fixed strategies, a workload adaptive strategy considers the recent workload in order to set a provisioning target. We represent the demand of the system as the number of compute nodes requested by the query plan at a second-by-second granularity. Note that this is not the same as CPU or other resource utilization but a record of the total resource requests of the workload. An example of a workload adaptive strategy is to set the target capacity of resources as the mean of the resource demand over the previous ten minutes. These strategies have the advantage that they are able to adapt as workloads change over time, more closely matching the workload demand as it changes. However, workload-adaptive strategies are insensitive to cost and execution environment. Changes in the cost of virtual machines or the elastic pool have no impact on their target. If the cost of virtual machines were to increase sharply,

an ideal strategy should use more of the elastic pool as the relative cost has decreased. Similarly, if the startup time of virtual machines changes, the strategy will not adapt to decrease cost.

Note that there is a difference between these strategies and those that are currently used by most auto-scaling systems. Because systems like Redshift [20], Databricks [16], or Snowflake [15] delay work rather than use an elastic pool, they do not make allocation decisions directly based on this demand history. Rather, they begin with a fixed set of resources and provision new nodes only when work begins to queue.

Similar to the fixed strategies that cannot adapt to the workload, these workload-adaptive strategies do not adapt to changes in environment. However, some configurations may be better adapted to some environmental conditions than others.

4.4 Dynamic Cost-Based Strategy

No single workload-adaptive strategy is able to minimize cost across all execution environments. The cost of any given strategy will depend on the relative costs of VMs and the elastic pool, and other factors including the startup time of VMs. Instead of choosing a single approach, we propose a cost-based meta-strategy that chooses the least costly of a family of workload-adaptive strategies given the observed workload and execution conditions. We require a way to determine what the cost of each *would have been*. This allows us to naturally adapt to varying conditions, such as changes in price or startup times, without fixing them as parameters of our model. In the remainder of this section we describe how our meta-strategy accomplishes this. We begin by describing how we collect a history of our workload.

4.4.1 Workload History. We collect a history of the number of tasks requested by running queries at a second-by-second granularity. This number increases when tasks are scheduled, and decreases when tasks complete. We assume that each task can run either on a VM or in the elastic pool. We also assume that the latency of this task is not dependent on this decision. We record the maximum number of concurrent tasks each second for the duration of our workload and call this the *workload history*.

4.4.2 Target and Allocation Histories. Each workload adaptive strategy takes the workload history as input and produces a target allocation of VMs. We record this allocation in a *target history* for each strategy. Assuming that we know the time required to start a VM (the latency between the request and the moment it is available to execute tasks), we can use this history to make an accurate prediction of what the number of VMs *would have been available* over the course of the workload had we chosen this strategy. We call this prediction the *allocation history*.

4.4.3 Cost Calculation. We determine each strategy's predicted cost over the *workload history* using the predicted *allocation history* of each strategy, and the cost model (historical prices) of both VMs and the elastic pool. We can generate an accurate prediction what the cost of each strategy *would have been* had we chosen it from the beginning of the workload. We assume that any time the *workload history* is less than the *allocation history* tasks would have executed on virtual machines. Conversely, when the *workload history* exceeds the *allocation history*, the excessive tasks would have been executed on the elastic pool. We can predict accurately the cost of this strategy over the course of the workload history because the cost of VMs and the elastic pool are known from historical data.

If environmental or cost conditions change, we can recompute each strategy's *allocation history* and determine the cost of the strategy under the new conditions with the above method.

4.4.4 Meta-Strategy Overview. We now have a way to determine the cost of different strategies given a history, cost models, and environmental conditions. What remains is to select a strategy from our family. We employ multiplicative weights [9] to make a choice among our family of strategies.

Multiplicative weights is a randomized algorithm for choosing among a family of "experts" given cost feedback over time. It provides a guarantee that the algorithm overall has an expected cost that is at most an additive factor over the best expert in the family. The algorithm does this by maintaining a weight for each expert and at each time step randomly choosing a expert based on the distribution of these weights. At each time step T it updates each expert's weight according to the cost incurred by the expert in the preceding interval. Specifically, it multiplies the weights by a factor $w^{T,1} = w^{t1}1 \quad \epsilon C^0$, where C is the normalized cost, or penalty, of each expert in the preceding interval. The algorithm then chooses an expert at random using a distribution according to the weights set above. Multiplicative weights guarantees that the overall strategy is within a factor of $\rho ln^1 n^0 \bullet \epsilon$ where ρ of the best performing expert where ρ is the maximum cost incurred per round, n is the number of experts, and $\epsilon <= 1 \bullet 2$.

In our case, each "expert" is a member of our family of strategies and the cost incurred is simply the cost incurred by each strategy over the preceding time period, as measured by our cost calculation.

A brief summary of our strategy follows.

(1) Generate a target from each strategy in our family given the *workload history* and record its *target history*.
(2) Given the observed VM startup time and the *target history*, generate an *allocation history* for each strategy.
(3) With the cost of VMs and the elastic pool, generate the cost of each strategy for the *workload history* with the *target history*
(4) Update the weights using the cost of each strategy.
(5) Randomly choose the current allocation *target* using the distribution of weights in the preceding step.

Thus we guarantee that we choose a strategy that is close to the best in the family. We execute the above meta-strategy at regular intervals throughout the workload, every five seconds in Cackle. However, the efficacy of this meta-strategy also depends on the strategies available.

4.4.5 Choice of Strategy Family. For the above meta strategy to be effective at reducing cost, we need to ensure that our family of strategies can produce a low cost for a wide range of conditions. While any number of strategies could be effective, there are some we should include to ensure it adapts well to a wide range of workloads. For instance, we should include a strategy that will set a target higher than any workload history seen in the *workload history*. This is to handle increasing workloads. For instance, in a *workload history* increasing linearly with time, the strategy that will minimize

cost must set a target number of VMs with more capacity than what is seen in the history to account for the delay in starting VMs.

While we could consider a set of strategies that attempts to predict the workload, we have found that a simpler set of strategies is sufficient to significantly reduce cost. In particular we use a set of "percentile" strategies with varying parameters. Each strategy has three parameters: a "lookback" defined in seconds, a percentile, and a multiplier. Each strategy takes as input the *workload history* and calculates the given percentile over the last "lookback" seconds of the history and multiplies it by the multiplier. We include in set of strategies, percentiles from one to one hundred each with a multiplier of 1.0, and 80th percentile strategy with multipliers ranging from 1.1 to 20 to allow for provisioning more than what we see in the history. For each of these parameters settings we include a strategy with a lookback ranging from 10 seconds to an hour. Thus, we have several hundred strategies in our family. We show the robustness of our meta-strategy with this technique through an analytical model in Section 5.

4.4.6 Cold Start Workload Considerations. At the beginning of the workload, the meta strategy has no way to effectively differentiate between strategies. This is less of a problem for Cackle than others as tasks will simply execute on the elastic pool if VMs are not available. As the history is growing in size during the initial stages of the workload, the meta-strategy may fluctuate between several strategies rapidly. This is reasonable as it is unclear what the workload will look like given the limited amount of data available. This means that the first part of the workload may suffer from higher cost than is optimal. One way to avoid this could be to add an expected workload to the history to prime the meta-strategy. We find that with relatively small VM startup times the overall effect of this excessive cost is small. As the history grows, we find that the meta-strategy typically settles on a single strategy as long as the environmental conditions and costs remain the same.

5 STRATEGY ROBUSTNESS

Cackle is intended to execute a query workload at minimal cost for a wide range of workloads and environmental conditions. For these analytical query workloads, environmental conditions include the time to start a new virtual machine, the cost premium of compute from the elastic pool, and the minimum billing time of workers in the elastic pool. In this section we describe our strategy for changing allocation based on these factors, build an analytical model to test our strategies for a range of workloads, and explore how our strategy compares to fixed, workload-adaptive, and an oracle strategy. We also demonstrate how the ability to access an elastic pool of resources to execute queries improves query latency with a minimum additional cost.

5.1 Model Implementation

We generate a workload of queries arriving in a fixed window. The start time of each query is chosen according to a distribution. Some queries will be uniformly distributed in the fixed window. The proportion of these queries is the `baseline load`. The remaining query start times are selected at random from a sine distribution.The goal of this distribution is to match workloads that we see in practice, with cyclical workloads and unexpected demand spikes. In particular, as we saw in in the startup and Alibaba 2018 workloads in Figure 2 and Figure 3, queries arrive at unexpected times but with some predictability in the rate of their arrivals.

Default Workload Parameters	
Workload Duration	12 Hours
# Queries	16384
Baseline Load	30%
Period Of Query Arrivals	3 Hours

Default Environment Parameters	
VM Startup Latency	3 Minutes
Minimum VM Billing Time	1 Minute
Cost of VM (2vCPUs)	$0.03/Hour
Cost of Elastic Pool (2vCPUs)	$0.18/Hour

Table 1: Default Workload and Environment Parameters for our Analytical Model

The model tracks resource demands at a second-by-second granularity. It measures the number of concurrently running tasks, the volume of data exchanged, and the number of messages between tasks. It assigns this demand first to available provisioned resources; any remaining demand is serviced by the elastic pool. The total cost of the workload is calculated based on the runtime of the provisioned resources and the use of elastic pools according to each respective cost model. For instance, provisioned virtual machines are charged by the second and have a minimum billing time of one minute, while reads and writes to cloud storage that are used for shuffling are charged by the request. Our cost models are based on the corresponding AWS [7] services where Cackle is implemented but could be trivially swapped out for other cost models.

We report the default values the analytical model uses for workload generation and the environment. in Table 1

We model the execution of each TPC-H [14] query with scale factor 100. Data used to model each query is taken from an execution of each query in our implementation on elastic pools, i.e., on AWS Lambda with shuffling through Amazon S3. We execute each query five times and collect runtime statistics from the execution of the query with median runtime. We collect the duration of each task, the dependencies between stages, the number of reads and writes to cloud storage, and the size of data shuffled. We measure metrics on shuffling to account for the cost associated with the shuffling layer. Duration of tasks is rounded to the nearest second with a minimum of one second.

We compare our dynamic strategy described in the previous section against a set of alternative strategies. We label our strategy `dynamic`. We compare against a set of fixed strategies that we label `fixed_x` where `x` is the number of virtual machines allocated for the duration of the workload. Therefore `fixed_0` corresponds to all tasks executing on the elastic pool. We also compare with workload-adaptive strategies. We use a mean strategy that takes the mean of the previous five minutes of history and multiplies with a constant `y` to get a target allocation, labeled `mean_y`. We also include a simple predictive model,`predictive`, that computes a linear regression over the previous five minutes, and sets a target equal to the maximum of the predicted workload after the VM startup time i.e., it attempts to predict the workload demand at the time VMs are started. Finally, we compare against an oracle strategy

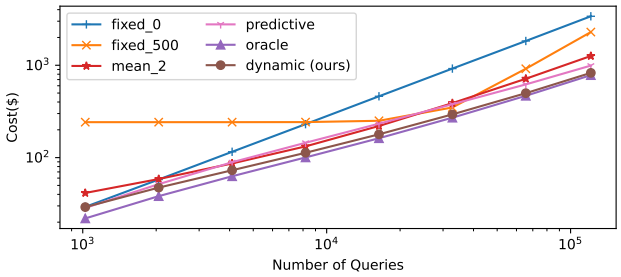

Figure 5: Cost of query workload, varying the number of queries in the workload.

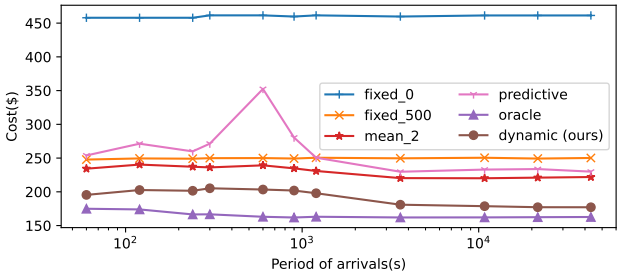

Figure 6: Cost of query workload, varying the period of query arrival times.

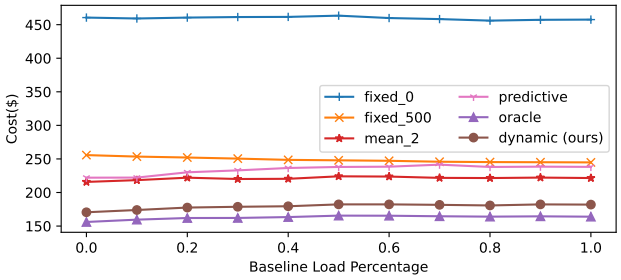

Figure 7: Cost of query workload, varying the baseline load of query arrival times.

with full knowledge of the upcoming workload that allocates the number of provisioned instances in order to minimize the cost of compute. It does not make modifications to the DAG of stages in the query execution but instead takes the resource demand curve as is minimizes cost only changing allocation. We label this strategy `oracle`.

5.2 Compute Workload Changes

We now demonstrate how the dynamic strategy is robust to workload changes. We vary the number of queries, the period of query arrivals and the baseline load of queries. The goal of our strategy is to be as close to the `oracle` strategy as possible over a range of workloads. To enhance clarity in the figures, we leave off strategies that have similar properties to others in each plot.

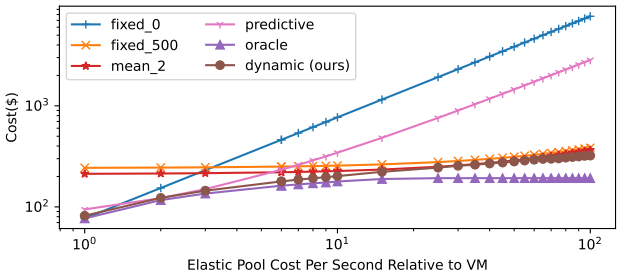

Figure 8: Cost of Query workload, varying the cost of Lambda relative to VM Cost.

5.2.1 ery Density. Figure 5 shows how the cost of strategies changes with the number of queries in the workload. The Oracle strategy outperforms all other methods when the number of queries is low as it has perfect knowledge of when queries will arrive. As the number of queries increases, the dynamic and mean strategies converge as the workload becomes more regular. The fixed workloads have constant cost until the number of queries exceeds their capacity and their cost grows. Their inability to adjust makes them an expensive choice unless the actual demand closely matches the fixed provisioning. For instance, `fixed_500` approaches the cost of `oracle` around 12,000 queries. This result indicates that simply augmenting a fixed provisioning with an elastic pool will achieve elasticity but at an exorbitant cost. The left portion of the plot demonstrates that fixed provisioning can cost several times as much as an elastic system over the same workload. Conversely, a system exclusively using an elastic pool, namely `fixed_0` can be an order of magnitude more expensive than a system that is augmented with provisioned instances when queries arrive frequently. The `dynamic` strategy is the lowest cost across the range excluding `oracle`, but the `predictive` strategy also performs well.

5.2.2 Period of Arrivals. Figure 6 shows how the cost of the system changes for each strategy as the period of query arrivals changes. Query arrival rates often peak daily as our example workloads in Section 2 show. But workloads can also include other frequencies of query arrival times. We see this in the startup workload in Figure 2 where a similar set of queries also arrives every 15 minutes. The fixed strategies see little impact of this change while the periodic strategy performs exceptionally poorly then the period wer are training on coincides with the periods of arrivals resulting in a larges spike. Because `dynamic` uses a family of strategies, it is less subject to these effects and performs the best among tested strategies, except the oracle.

5.2.3 Baseline Load. Figure 7 shows how the cost of various strategies are impacted by making queries arrive more uniformly. As the arrival rate of queries evens out and fewer queries exceed the capacity of the `fixed_100` allocation the cost decreases, this is true to a lesser extent for `fixed_500` also. However, other strategies see little impact as the workload shifts.

5.3 Environment Changes

In addition to workload changes, the execution environment may change with the same workload. One example is the relative price

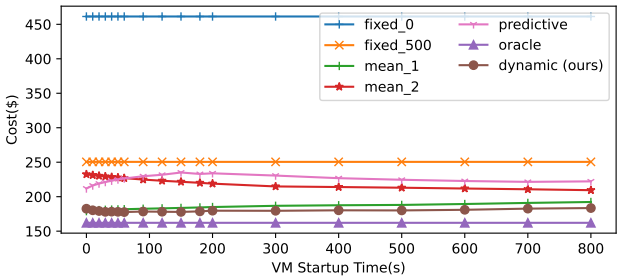

Figure 9: Cost of query workload, varying the startup time of virtual machines

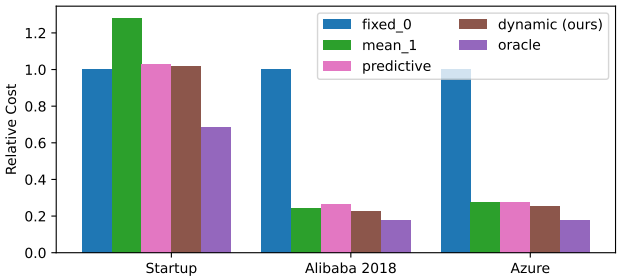

Figure 10: Cost of workloads, varying the startup time of virtual machines. Normalized to the cost of fixed_0

of provisioned resources to the elastic pool. An increase in the cost of a VM or the decrease in the cost of the elastic pool can cause the cost premium of the elastic pool to change. This is common. In the period from Jan 1 to March 31, 2023 the spot price of a `c5a.large` instance with 2vCPUs and 4GB of DRAM increased from $0.0343 to $0.0670 per hour, nearly doubling. The cost of AWS lambda for a similar instance was fixed over the same period, making the relative cost difference decrease from 7x to 3.6x the spot price of VM. All other factors being equal, a sound strategy should respond to this change by using the elastic pool more frequently.

5.3.1 Elastic Pool Costs. We show the impact of varying elastic pool cost in Figure 8. If the elastic pool has equivalent cost to a provisioned instance, it is never worthwhile to start a long-lived virtual machine (unless there are caching benefits or other significant performance gains). Thus, `fixed_0`, running only on the elastic pool is tied for the least expensive. Cost insensitive strategies like `fixed_500` and `mean_2` are significantly more costly when relative cost of the elastic pool is close to the VM cost. As the cost premium of the elastic pool grows relative to VMs, it is less costly to provision VMs to execute some portion of tasks. The lower cost of `dynamic` compared to `fixed_0` and `fixed_500` shows the risk of strategies that either use exclusively VMs or the elastic pool. Our `dynamic` strategy is significantly less expensive than both. The `predictive` strategy performs well until the elastic pool increases in price. This shows the risk of not considering changing costs when choosing a strategy. Our `dynamic` strategy remains close in cost to the `oracle` until elastic pool costs grow to more than 10x the VM. After this the cost of any work executing on the elastic pool causes all but the `oracle` strategy to grow rapidly.

5.3.2 Virtual Machine Start Time. Figure 9 shows the impact of the latency of starting virtual machines on the cost of various strategies. It is reasonable to expect that this might change over time as improvements are made by cloud service providers. In our experiments we find that VMs take around three minutes from their request to run on AWS. Fixed strategies are, of course, unaffected by this change as they never start or stop virtual machines. Similarly, the `oracle` strategy is unaffected as it knows exactly when instances should be started to service a given demand and this is unaffected by how long the provisioned instance takes to start. However, it does have meaningful impact on the mean strategies. `mean_2` is significantly more expensive than `mean_1` when provisioned instances are fast to start but much less expensive when

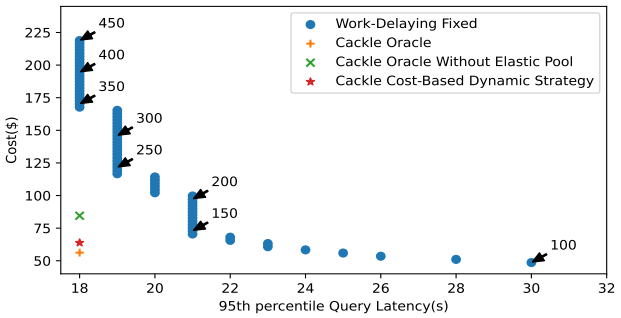

Figure 11: Cost and latency of delaying strategies with fixed provisioning vs Elastic Pool strategies. Each blue dot shows a fixed provisioning with a different number of VMs, marked regularly with arrows and labels.

virtual machines are very slow to start. `dynamic` is much closer optimal for most startup times.

5.4 Example Real Workloads

We next compare the performance of different strategies on a set of real world workloads we introduced in Section 2. As we do not have access to the exact performance characteristics of these datasets we made a series of assumptions. For the workload from the startup, we had access only to start and end times of queries. We made the simplifying assumption that each time a query starts it selects a random TPC-H query at scale factor 100 and we proceed as we have for the previous sections. In the Azure workload, we we given how many "nodes" were requested at each time interval. Here we assume that each "node" requested was the equivalent of 20 tasks running, we then used this demand curve in our analytical model. Finally, for the Azure synapse workload we had the number of CPUs requested by all queries in the system. We assumed that one CPU is equivalent to a single task, used this to generate a demand curve, and proceeded using our analytical model. The results are shown in Figure 10. We normalize to the cost of the fixed_0 strategy. Besides the oracle, our dynamic strategy is either the best or within 1% of the best strategy.

5.5 The Cost of Delaying Work

Unlike Cackle, OLAP systems typically schedule work until the system is fully utilized. Any additional work will be queued until system resources are available either due to prior work completing or new resources coming online. Thus, we consider the cost and latency impact of delaying work. We model a workload delaying system and demonstrate how Cackle is able to reach new points in the space for some workloads. We use a query workload of 2048 queries over 12 hours with a baseline load of 30% and a period of 12 hours. We show the latency and cost of this workload on a work delaying system with fixed provisioning. We show a range of fixed provisionings with blue dots, regularly labeling the fixed number of VMs in Figure 11. In addition we compare an oracle strategy that always ensures there are enough VMs to execute all available tasks in workload as soon as they arrive, but does not have access to an elastic pool. We label this strategy `Cackle Oracle Without Elastic Pool`. Finally, we compare against an oracle strategy with access to an elastic pool, labeled `Cackle Oracle` and our dynamic strategy with an elastic pool (`Cackle Cost-Based Dynamic Strategy`). Tasks are scheduled on a first in first out order with priority given to the earliest submitted queries. Figure 11 shows the cost and 95th percentile latency of queries for these strategies. No fixed strategy approaches the bottom left region. That is, fixed strategies are unable to achieve the latency of an over-provisioned system at the cost of an under-provisioned system. Furthermore, our cost-based dynamic strategy is able to achieve the same latency at a lower cost than the work delaying system *even when the work delaying system has perfect workload knowledge*. This is because the cost model of virtual machines requires a minimum billing time of 60 seconds while the elastic pool allows for fine grained billing. For short bursts of compute the cost premium of the elastic pool is less expensive than paying for the minimum billing duration of the same number of virtual machines.

5.6 Shuffling

We model the shuffling layer in a similar way to the compute layer. The primary difference between modeling the cost of the shuffling subsystem and the execution layer is that individual reads and writes to caching nodes and the elastic pool are tracked for maximum fidelity to the implementation we describe below. We do this to account for the difference in the cost model of provisioned shuffling nodes and cloud storage. Specifically, the shuffling nodes have limited memory to store intermediate state and are charged by the minute, while cloud object storage is billed per request and has effectively unlimited storage. Because of the large cost of the individual reads and writes to cloud storage, it is almost always less expensive to over-provision VMs for the shuffling layer. For this reason, instead of our cost-based strategy, we instead choose a target number of shuffling nodes with sufficient memory to hold the maximum amount of intermediate state seen in the workload in the last 20 minutes. In addition, we never set a target of less than 16GB of available shuffling node memory to ensure that some shuffling nodes are always available to decrease the number of read and write requests to cloud object storage.

6 DISCUSSION

In this section we describe the core assumptions and requirements required for a system to integrate Cackle's techniques. At it's core, Cackle is a method for deciding when to use an elastic pool of resources that has close to equivalent performance to a provisioned resource. While other execution environments may or may not require a shuffling layer, or use a cloud function service directly, they can still use the core technique. In particular many data warehousing providers keep a "warm pool" of nodes for their customers or use a set of pre-allocated nodes for "serverless" workloads to be executed on demand. These pools may use identical hardware to provisioned instances, but are more expensive because they are pre-provisioned to meet a rapid demand. Given the complexity of this environment, including variability in the startup time of resources, and the cost difference of the elastic pool and provisioned resources, Cackle provides a method to choose the a split of provisioned and elastic resources that minimizes workload cost.

Cackle requires the elastic pool to be immediately available to react to a change in demand. Beyond this we assume only pricing for the elastic pool and provisioned resources, as well as environmental information including how long provisioned resources require to start. While we do not currently consider the performance differences that might arise due to data locality, we believe that Cackle could be extended in future work to account for these concerns.

7 EVALUATION

We evaluate Cackle by first ensuring that our analytical model produces accurate results when compared to a real execution of the same workload. We also ensure that the costs of executing workloads on Cackle is within a reasonable margin of the cost of an oracle strategy. Second, we compare the cost and performance behavior on workloads with a varying number of queries to both a modern data warehouse, Databricks SQL with fixed provisioning and with autoscaling enable. We will show that Cackle achieves significantly improved cost and performance stability across different workloads compared to Databricks SQL.

7.1 Experimental Setup

The general outline of Cackle is given in Section 3, here we describe the implementation details of our system. All experiments were conducted on AWS in the `us-east-1` region in a single availability zone. We believe our techniques generalize and could be adapted to other systems, we implemented our system using Starling [27] using code shared with us by the authors. We considered implementing our techniques on Spark [32], but we found that it took tens of seconds to start a Spark Executor on AWS Lambda, our elastic compute pool. This violated the first constraint on our elastic pool we described in subsection 2.2 and made it a nonviable choice. With a pre-warmed pool of spark workers, it should be feasible to implement the same approach.

7.1.1 Base Tables. We store our base tables are Amazon S3 [6] in ORC [25] format. We first generate the data with the TPC-H [14] data generator in 100MB chunks. We then convert these files to ORC format. We do not partition tables on join keys or sort data.

7.1.2 Compute Layer. The compute layer of Cackle is similar to Starling [27] with some key differences. Instead of running exclusively on AWS Lambda, tasks can also be run on provisioned virtual machines. To achieve this, VMs in the compute layer run a server that receives requests to execute tasks. Because these virtual machines and the Lambda functions must be able to exchange data through the shuffling layer, we ensure that they are provisioned in the same availability zone. AWS Lambda is configured to run functions in the same virtual private cloud, availability zone and security group as the other components of the system and uses a memory size of 3GB at a cost of $0.18 per hour.

Instances are provisioned using spot instance requests to reduce overall system costs. They are restricted to a single availability zone to eliminate the excessive cost of cross AZ traffic. The request specifies 2vCPU instances with at least 4GB of DRAM with some instance types excluded due to low network throughput. To change the number of provisioned instances the coordinator submits a modification to the spot request to change the number of instances. When a VM starts up it connects to the coordinator to signal its availability. Afterwords, it accepts work until it is terminated.

We find that in our experiments 99% of lambdas start within 200ms. This satisfies our requirement for rapid availability we need to implement Cackle

While our algorithm assumes that the performance of tasks will be equal independent of where they are scheduled, we find that VMs execute tasks 25% faster than on AWS Lambda. This is because the actual instances of VMs are decided by the current spot prices and may be more performant than the equivalent Lambda instance. Similarly, shuffling through our caching layer results in query performance 15% faster than without when the cache is significantly over-provisioned. Of course the load on all components will impact these results. Despite the divergence from our algorithmic assumptions, we find that our approach still achieves the results comparable to what we expect.

Tasks in the compute layer are given a list of available shuffling nodes when executed. When a task needs to shuffle data to a successive stage, it first attempts to write to the available shuffling nodes. If full, the task instead writes its data to S3 using the same technique as Starling and Lambada.

7.1.3 Shu ling. Distributed analytical query execution requires compute nodes to exchange intermediate state in an all-to-all shuffle. This occurs most frequently to execute distributed joins. While other data warehouses shuffle data directly between the nodes in the system [8], Cackle is not able to do so for two reasons. First, the elastic pool we use in our compute layer cannot exchange data directly, as the AWS Lambda disallows inbound connection requests. Second, the aggregate memory in shuffle VMs may not be enough to hold intermediate state when demand peaks. The shuffling layer is needed to mitigate this limitation of our elastic pool.

Thus, we implement a distinct shuffling layer that temporarily stores intermediate state. Like the compute layer, we use both a set of provisioned resources and an elastic pool. As AWS Lambda is unable to exchange data directly, we instead use S3[6] as an elastic pool of compute. Starling [27] and Lambda [24] use S3 exclusively. But for workloads with a larger number of queries, the cost of using S3 can dominate the overall workload cost. This is because S3 charges users by the number of PUT and GET requests. These prices are high enough that it can account for half the cost of the cost of query execution [27]. For a workload with many queries and therefore many read and write requests, it is less expensive to provision dedicated shuffling nodes. Under Starling and Lambada's method of shuffling a single 128 task to 128 task shuffle requires 256 S3 PUTS and 128*128*2 S3 GET request, costing $0.013. At this price, we could instead run a dedicated VM shuffle data with 8GB of memory for ten minutes at the spot price of $0.08 per hour (the Feb 2023 price of a c5.xlarge instance). For most workloads adding shuffling nodes will quickly reduce cost compared to the elastic pool of S3. This we also use a set of provisioned VMs to reduce this shuffling cost, and fall back on using S3 only when provisioned VMs are at their capacity.

Shuffling instances act as an in-memory key-value store and are implemented as a gRPC [19] server. Shuffle partitions destined for the same partition are returned in a single request.

Shuffling instances are provisioned with AWS's spot requests, requiring nodes with 4vCPUs and at least 8GB of DRAM. Again, some instance types with low network throughput are excluded. This is especially important for shuffling as we require large network bandwidth to exchange data with the compute layer.

When invoking each task in a query, we pass a list of the available shuffling nodes. All tasks in the query are given the same list of shuffling nodes. Each task attempts to write its partitioned output first to a shuffling node. Shuffle nodes for a write are chosen using a hash of the destination task of the partition. If a shuffle node is full, the partition tries two more nodes before falling back on S3.

While there is a significant amount of work on shuffling in this context [21, 22, 28]. , we instead chose a simple caching layer + S3 approach to avoid the complexity of these approaches as shuffling did not appear to be a large driver of cost.

7.1.4 ery Plans and Execution. We generate C++ code for all queries from the physical execution plan. This code is compiled and uploaded to AWS Lambda where it can be invoked with necessary parameters for execution, including a unique query identifier, stage and task numbers and the current set of shuffle nodes. The code for each query is also stored on S3 where it is fetched by VMs in the execution layer for execution. The physical plan for these queries were borrowed as closely as possible from the planner of Redshift [20], except that all joins are either broadcast joins or partitioned hash joins. The number of tasks in each stage is hand tuned for performance and to ensure that memory requirements for each task do not exceed limits of a Lambda function invocation or VM. Stages are scheduled as soon as their dependencies are resolved. We disable pipelining between stages.

7.1.5 The Coordinator. The coordinator is responsible for receiving requests to run queries, scheduling tasks on the compute layer, and allocating and keeping track of provisioned resources. It runs on a single on-demand `c5a.xlarge` instance at a cost of $0.154 per hour. For our experiments the coordinator receives a schedule of queries to execute as a list of queries from the TPC-H benchmark and starts them at a predetermined time.

The coordinator keeps a *workload history* at the granularity of one second. This history includes the number of concurrent scheduled tasks, and the volume of data shuffled. Resources are

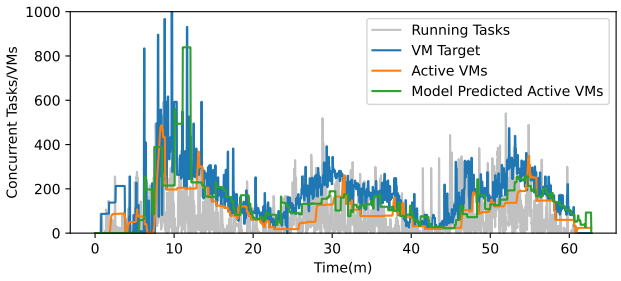

Figure 12: The demand for compute, the target number of VMs, the number of running VMs, and the number of Available VMs our analytical model predicted over the duration of an hour long workload of 750 queries.

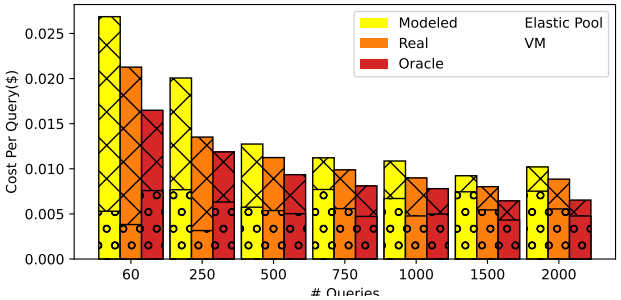

Figure 13: Cost comparison between our analytical model, a real execution, and our model with an oracle strategy on a set of workloads. The lower portion(circles) shows the cost of VMs while the upper (crosses) shows the elastic pool cost.

allocated using the `dynamic` strategy described in Section 4. For the evaluation the allocation strategy assumes that the startup time of virtual machines is three minutes as this is the time we have experimentally observed. And the cost premium of AWS Lambda over spot instances is 6x. That is, a minute of compute on a lambda function costs six times as much as the equivalent spot instance.

7.1.6 ery Workloads. We use a set of hour-long workloads generated with our workload generator, also used by our analytical model. We use a mixture of TPC-H and TPC-DS queries over scale factor 10, 50, and 100 datasets. We include all queries from TPC-H and three from TPC-DS to achieve a broader set of queries than TPC-DS alone. In particular we add TPC-DS queries 24, 58, and 81 because they are an iterative query, a reporting query, and a query with multiple fact tables. Our generator creates a workload of queries with a 30% baseline load and a period of 20 minutes for the remaining queries in a sinusoidal distribution. Thus, each workload will have peak in the density of queries at 10, 30 and 50 minutes. Each query is randomly selected uniformly from the set and scale factors. We compare workloads with varying numbers of queries, from 60 to 2,000 queries in hour-long duration.

7.1.7 Databricks SQL. We compare Cackle with a Databricks SQL Serverless Warehouse [16] with several configurations in AWS. We first load all data into Databricks. Before executing each workload, we first execute all queries in the workload three times. This allows the data warehouse to populate its local disk caches and warm up. Note that this makes for a somewhat unfair comparison with Cackle. We execute queries from a virtual machine on EC2 and we measure time of queries from the time each query is submitted until results are returned this includes. We report cost by measuring the time virtual machines are running. Assuming that Databricks SQL instances run on the same hardware as their provisioned instances, the instance types are `i3.2xlarge`, `i3.4xlarge`, and `i3.8xlarge`. `i3.4xlarge` and `i3.8xlarge` are used for driver instances and use on-demand. The `i3.2xlarge` are used as worker instances.

We compare with four configurations of Databricks SQL. First, we consider fixed size warehouse of five `small` clusters, each with one `i3.4xlarge` driver and four `i3.2xlarge`. We also compare with a fixed size `medium` warehouse with three clusters, each with one `i3.8xlarge` driver and eight `i3.2xlarge` worker nodes. We also consider autoscaling `small` and `medium` warehouses that begin with one cluster can scale to eight and five clusters respectively. We report the cost of Databrickss as $0.70 per DBU(Databrick's charging metric) per hour. Each Small cluster is 12 DBU and each medium cluster is 24 DBU.

7.1.8 Redshi Serverless. We also compare against a Redshift Serverless instance set up with a base capacity of 8 RPUs(Redshift's internal sizing metric). Each RPU costs $0.36 per hour. Thus the cost of Redshift Serverless is $2.88 per hour. Users are charged only when queries are running on the cluster with a minimum charge of 60 seconds. Redshift Serverless instances allow users to increase the capacity they use when usage is high and will allocate additional hardware (at an additional charge). Again, we first load all data and before executing each workload, we execute all queries in the workload three times to allow caches to populate.

We believe that the techniques and behaviors we see in Databricks SQL and Redshift Auto-scaling are representative of the techniques employed by other data warehouse products. Specifically, Snowflake Auto-Scaling Multi Cluster warehouses [29] wait until work has queued before allocating new hardware to a user's workload.

7.2 Analytical Model Validation

We first validate that our analytical model matches the behavior of Cackle during a real execution. Figure 12 shows the *compute* resource demand execution of an hour long workload of 750 queries. We see that the demand rises and falls with a twenty-minute period. Cackle adjusts the target number of virtual machines, seen in blue. The number of virtual machines that are active (either running tasks or available to run tasks) is shown in orange. We took the executed history from the workload and fed it back into our analytical model to validate our cost prediction was accurate. We show the results as "Model Predicted Active VMs" in Figure 12 as well. Our analytical model predicts that for this workload the cost of the compute layer should be $8.41. The actual costs measured by the system for the execution of this workload were $7.41, a difference of 12% compared to the prediction of the analytical model. The predicted cost of the elastic pool and virtual machines were $2.63 and $5.07, respectively, while the actual costs measured were $3.22 and $4.19.

In Figure 13 we show the predicted and actual compute costs for a set of workloads of with varying numbers of queries. We also compare the costs of an oracle strategy on our analytical model for

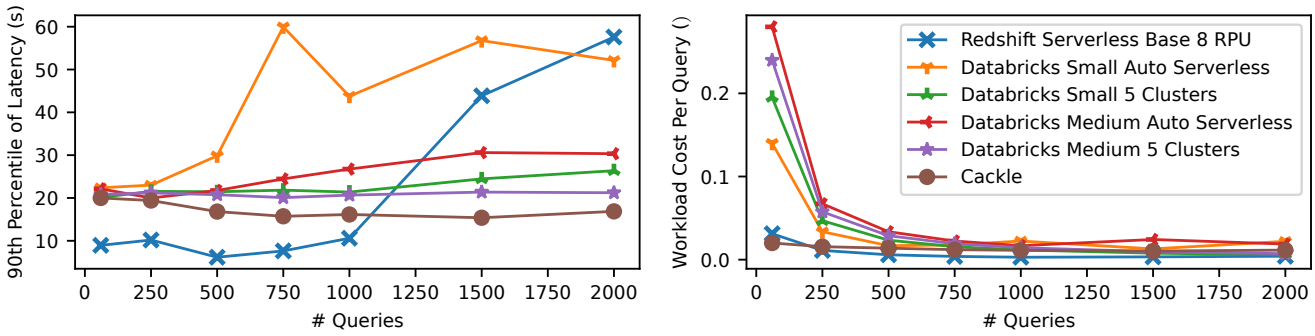

Figure 14: Comparison of Cackle query latency and cost with Databricks SQL `small` and `medium` warehouse fixed provisioning and with auto-scaling enabled, and Redshift Serverless.

comparison to the best-case provisioning. The oracle strategy has complete workload knowledge. We see that the cost our analytical model predicts is close to the real cost of executing the workload in all cases. Furthermore, we also see that the model also accurately predicts the cost ratio between the elastic pool and provisioned VMs. Finally, we see that for workloads with a small number of queries, the majority of the cost comes from the use of the elastic pool, even using the oracle strategy. For busier workloads this percentage decreases significantly. We find that for this set of workloads our system is a maximum of 36% more expensive than a strategy with full workload knowledge. Most of this cost reduction comes from the ability of an oracle strategy to know the arrival time of demand spikes and provision VMs to meet them. The average increase in cost compared to the oracle strategy is 26%. Despite this higher cost per query, we show below how the system has lower overall cost than alternatives for low query workloads, and maintains cost and performance stability across a range of workloads.

7.3 Cost and Latency Stability

In Figure 14 we compare the cost per query and the 90th percentile of query latency against four configurations of a Databricks SQL Pro Warehouse and Redshift Serverless. We use the 90th percentile of queries to capture the user experience of the system, specifically including when the system is under heavy load. We compare a fixed provisioning and an auto-scaling strategy on the small and medium Databricks SQL warehouses. On the left we show how the latency of queries changes as the number of queries increases. We find that the Databricks auto-scaling strategies have relatively low latency when the number of queries is low, but as we increase the number of queries the 90th percentile latency of queries can increase by more than 2x for Small Autoscaling Clusters. This is expected as queries queue while additional clusters are allocated. Similarly Redshift Serverless tail latency also worsens by more than 6x. The fixed provisioning also sees a reduced performance as the number of queries increase, but to a lesser extent. Cackle has consistent query latency across all workloads tested.

On the right side of Figure 14, we compare the per query cost of each configuration. We include both compute and shuffling costs for Cackle. We see that across a wide range of workloads, Cackle maintains a relatively stable cost per query. However for all other configurations, with the exception of Redshift, we see costs rise sharply as the number of queries in the workload decreases. This is because for these workloads we are paying for underutilized resources that remain mostly idle.Redshift does not charge a user for the time when queries are not executing, however, it appears that these nodes are still running in the background and thus will continue to incur a cost to the cloud provider. We note that the workload cost is lower for the Auto-scaling configurations for workloads between 200 and 800, but this comes with increased query latency. We see that the cost per query increases for these auto-scaling systems, because they do not rapidly respond to variations in query load, as Cackle does.

Cackle does not achieve as low of a cost as Databricks for larger workloads, or as low as Redshift across the spectrum. There are a few reasons for this. First, Cackle is a research prototype and performance and cost could both be improved with additional engineering effort. Cackle, unlike Databricks and Redshift does not use local fast disks to store base tables, and must exchange data through an additional shuffling layer. This is due to the constraints of the elastic pool used, AWS Lambda, and contributes both to cost and query latency. We believe that in a full integration of an elastic pool with a modern data warehouse could drive down cost and maintain high performance across the full range of workloads.

8 RELATED WORK

There is prior work on using elastic pools for query execution. Both Starling [27] and Lambada [24] proposed using Cloud Functions for query execution, but are only cost effective when query volumes are low. This approach is viable for a narrow slice of workloads. Instead, Cackle focuses on augmenting provisioned resources with elastic pools to gain elasticity without sacrificing the low cost of provisioned systems. In addition, our work proposes strategies for changing workloads and environment while both Starling and Lambada rely exclusively on elastic pools.

There is a wide body of work in adjusting the compute to changing demand in implemented in cloud systems. Redshift Concurrency Scaling [8] is one example of these strategies. Concurrency Scaling assigns new clusters to a database to handle spikes in load. However, this only occurs once queries have been queued on a user's main cluster. Databricks SQL has a similar technique for auto-scaling clusters, adding a cluster or more at a time **only after queries are queued** [1]. In addition to being slow to respond

to demand changes, these techniques also release added clusters slowly. Snowflake uses similar strategies [23, 29].

P-store [31] is an OLTP database that elastically scales compute resources to predicted changes in demand. However, it does not make use of elastic pools and instead relies on detecting workload changes and provisioning resources before demand spikes.

9 CONCLUSION

In this work we demonstrated that by augmenting a system with elastic pools of resources and choosing a resource provisioning strategy that accounts for cost and properties, we can avoid the common provisioning challenges common in data warehousing systems. We developed a novel meta-strategy that adapts to different workload and environmental conditions. We showed in an analytical model that this strategy is robust to varying workloads and execution environments. We validated the results of this model in an implementation and showed that our model predicted within 5% of the actual cost of the workload. Cackle achieves 90th percentile query latency an order of magnitude lower than a Databricks SQL medium warehouse with auto-scaling enabled with similar cost per query. Furthermore, it achieves lower variance in latency and performance than all compared configurations for a range of workloads.

REFERENCES

[1] 2023. Queueing and autoscaling. Databricks SQL Documentation. Posted at https://docs.databricks.com/sql/admin/create-sql-warehouse.html#queueing-and-autoscaling..

[2] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, Lazar Davidovic, Maja Damjanic, Slobodan Djunic, Nemanja Djurkic, Charles Feddersen, Cesar Galindo-Legaria, Alan Halverson, Milana Kovacevic, Nikola Kicovic, Goran Lukic, Djordje Maksimovic, Ana Manic, Nikola Markovic, Bosko Mihic, Ugljesa Milic, Marko Milojevic, Tapas Nayak, Milan Potocnik, Milos Radic, Bozidar Radivojevic, Srikumar Rangarajan, Milan Ruzic, Milan Simic, Marko Sosic, Igor Stanko, Maja Stikic, Sasa Stanojkov, Vukasin Stefanovic, Milos Sukovic, Aleksandar Tomic, Dragan Tomic, Steve Toscano, Djordje Trifunovic, Veljko Vasic, Tomer Verona, Aleksandar Vujic, Nikola Vujic, Marko Vukovic, and Marko Zivanovic. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3204–3216. https://doi.org/10.14778/3415478.3415545

[3] Alibaba Cluster Trace Program - Cluster Trace v2018 2018. Alibaba Cluster Trace Program - Cluster Trace v2018. Github. Posted at https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md..

[4] Amazon EC2 Spot Instances Pricing [n. d.]. Amazon EC2 Spot Instances Pricing. https://aws.amazon.com/ec2/spot/pricing/.

[5] Amazon Redshift Serverless [n. d.]. Amazon Redshift Serverless. https://aws.amazon.com/blogs/aws/introducing-amazon-redshift-serverless-run-analytics-at-any-scale-without-having-to-manage-infrastructure/.

[6] Amazon S3 [n. d.]. Amazon S3. https://aws.amazon.com/s3/.

[7] Amazon Web Services [n. d.]. Amazon Web Services. https://aws.amazon.com/.

[8] 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.

[9] Sanjeev Arora, Elad Hazan, and Satyen Kale. 2012. The multiplicative weights update method: a meta-algorithm and applications. *Theory of computing* 8, 1 (2012), 121–164.

[10] AWS Lambda [n. d.]. AWS Lambda. https://aws.amazon.com/lambda/.

[11] Azure Blob Storage [n. d.]. Azure Blob Storage. https://azure.microsoft.com/en-us/products/storage/blobs/.

[12] Azure Synapse Analytics [n. d.]. Azure Synapse Analytics. https://azure.microsoft.com/en-us/products/synapse-analytics/.

[13] Mengchu Cai, Martin Grund, Anurag Gupta, Fabian Nagel, Ippokratis Pandis, Yannis Papakonstantinou, and Michalis Petropoulos. 2018. Integrated Querying of SQL database data and S3 data in Amazon Redshift. *IEEE Data Eng. Bull.* 41, 2 (2018), 82–90. http://sites.computer.org/debull/A18june/p82.pdf

[14] Transaction Processing Performance Council. 2008. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html* 21 (2008), 592–603.

[15] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[16] Databricks SQL [n. d.]. Databricks SQL. https://www.databricks.com/product/databricks-sql.

[17] Google Cloud Functions [n. d.]. Google Cloud Functions. https://cloud.google.com/functions.

[18] Google Cloud Storage [n. d.]. Google Cloud Storage. https://cloud.google.com/storage.

[19] gRPC [n. d.]. gRPC. https://grpc.io/.

[20] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1917–1923. https://doi.org/10.1145/2723372.2742795

[21] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. 2022. Jiffy: Elastic Far-Memory for Stateful Serverless Analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 697–713. https://doi.org/10.1145/3492321.3527539

[22] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 427–444.

[23] Themis Melissaris, Kunal Nabar, Rares Radut, Samir Rehmtulla, Arthur Shi, Samartha Chandrashekar, and Ioannis Papapanagiotou. 2022. Elastic Cloud Services: Scaling Snowflake's Control Plane. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 142–157. https://doi.org/10.1145/3542929.3563483

[24] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 115–130.

[25] ORC Specification [n. d.]. ORC Specification. https://orc.apache.org/specification/.

[26] Ippokratis Pandis. 2021. The evolution of Amazon redshift. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3162–3174.

[27] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 131–141.

[28] Marc Sánchez-Artigas, Germán T. Eizaguirre, Gil Vernik, Lachlan Stuart, and Pedro García-López. 2020. Primula: A Practical Shuffle/Sort Operator for Serverless Computing. In *Proceedings of the 21st International Middleware Conference Industrial Track* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/3429357.3430522

[29] Snowflake Documentation: Multi-cluster Warehouses 2023. Multi-cluster Warehouses. Snowflake Documentation. Posted at https://docs.snowflake.com/en/user-guide/warehouses-multicluster#label-mcw-scaling-policies..

[30] Tableau [n. d.]. Tableau. https://www.tableau.com/.

[31] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-Store: An Elastic Database System with Predictive Provisioning. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 205–219. https://doi.org/10.1145/3183713.3190650

[32] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.