Serverless Query Processing on a Budget

William Ma
University of California, Berkeley
williamma@berkeley.edu

Jack Zhang
University of California, Berkeley
jack.tiger.zhang@berkeley.edu

ABSTRACT

Relational query processing is an ideal candidate for serverless computation with its stateless, idempotent, and short-lived properties. However, current serverless offerings for query processing neither provide millisecond-based pricing nor allow users to optimize the cost of their queries. To have reasonable trade offs for the cost of their queries, users are limited to using traditional serverful approaches. We demonstrate that serverless offerings can improve query execution times by up to 50% with no significant affect on the cost. To be able to support serverless offerings and give users the benefits of serverless offerings, we introduce a model that can estimate the query execution time given a trace of a previous execution of the query.

CCS CONCEPTS

• **Information systems → Query planning**; **MapReduce-based systems**.

KEYWORDS

Serverless, Query Planning, Spark

ACM Reference Format:
William Ma and Jack Zhang. 2021. Serverless Query Processing on a Budget. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Cloud computing achieved widespread adoption over the past decade, which has freed users from the burden of maintaining their own clusters and allowed them to provision their clusters on-demand. Similar to cloud computing, *serverless computation* has freed users from even provisioning their own clusters and allows them to achieve "web-scale" without a thought for how to optimally provision their clusters. As serverless computation gains momentum, service providers are introducing new services that leverage serverless, such as AWS Athena [2] and GCP BigQuery [4], which allow users to query their remote object stores without having to provision their own compute clusters.

Unlike newer serverless offerings (e.g., AWS Lambda [3], GCP Function [5]) that offer millisecond-based pricing, these services are priced on the amount of data accessed by the query, which is not optimal for the service provider or the user. The only way for the user to reduce the amount of data accessed is to approximate queries, which is not always desirable. Furthermore, pricing based on the amount of data accessed does not reflect the actual cost, which is the product of the wall-clock time, the cost of the compute node used, and the number of compute nodes used. Thus, the user pay the same amount for running two select query (e.g., `SELECT ... FROM TABLE_1` and `SELECT ... FROM TABLE_2`) and one join query (e.g., `SELECT ... FROM TABLE_1, TABLE_2`), as shown in table 1.

Query	Wall-Clock Time	Cost
2 SELECT statements	2 min	114 GB * $\frac{\$5}{1TB}$ = 0.72
1 CROSS PRODUCT statements	30+ min	114 GB * $\frac{\$5}{1TB}$ = 0.72

Table 1: Run time and cost of two sets of statements with GCP BigQuery. The fact that the two statements cost the same amount of money but have such drastically different run times is concerning.

Our key observation is that a pricing scheme based on wall-clock time will allow the service provider to have a fairer pricing scheme. This will also allow users to trade off run time for cost.

Although there are existing systems [7, 13, 16] that can, given a budget for run time or dollar cost, calculate the optimal provisioned cluster that optimizes for the other, these systems work only in a fixed-provisioned cluster throughout the query.

Additionally, there is an opportunity to expand the Pareto curve with serverless offerings. Although there are limits on how far the time-cost trade off curve can be pushed due to the maximum degrees of parallelism for the minimum run time and the optimal degrees of parallelism for minimum cost, serverless is able to find significant improvements in the run time and cost in other parts of the Pareto curve. For instance, figure 1 shows the stages of the execution graph of a Spark query. It is clear that there are three distinct stages of the Spark query that can benefit from a different number of nodes at each stage.

We propose a model based on Spark [19] that provides the optimal time-dollar cost tradeoff to the user and show preliminary results of a 50% run time reduction with only a 2% increase in cost from using serverless computation.

In order to develop on Spark, which is not serverless, we build our model based on a simulated version of Spark that is serverless, or *Serverless Spark*. In Serverless Spark, we assume that (i) we always have access to nodes ready to load data and execute tasks without having to warm up the JVM and (ii) we can have multiple Spark drivers running simultaneously to execute one query. We also

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

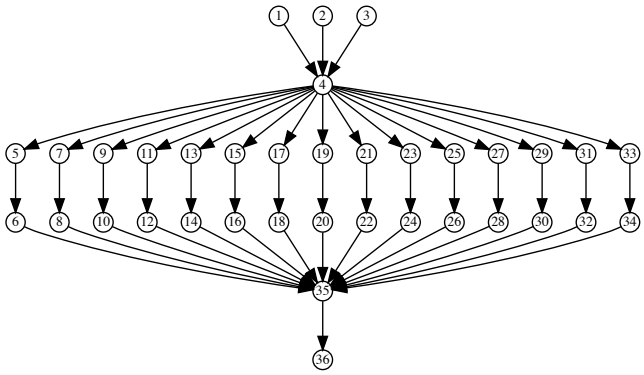


Figure 1: Spark execution graph for a sample TPC-DS query. Clearly, there are opportunities for large performance gains and cost savings with elasticity.

assume that the latency to launch new Spark drivers with the correct number of nodes attached is 125ms. These assumptions follow the standard of many current cloud services (e.g., AWS Lambda [3], GCP Function [5]) and, thus, are reasonable assumptions to make on cloud service providers.

Our contributions in this paper consist of

- A Spark simulator that can estimate the run time and cost of a query given a trace of a previous execution (section 2).
- Determine the time-cost trade off curve of a query (section 3.1.1).
- Optimize the run time or cost of a query given a cost or run time budget (section 3.1.2).

The rest of the paper is organized with discussion of the Spark Simulator in section 2, discussion of the Spark Serverless Simulator in section 3, presentation of our current results in section 4, related work in section 5, and future work in section 6.

2 SPARK SIMULATOR

Our Spark simulator simulates a simplified version of the Spark task scheduler. In Spark, each query is broken up into stages that form a DAG, such as the one shown in figure 1. Within each of these Spark stages, there are a number of tasks that can be executed independently of each other. With this understanding of Spark queries, we make a number of assumptions (section 2.1) to simulate the query on any serverless cluster provisioning (section 2.2). Because we are simulating on a trace of a previous execution, this leads to a number of sources of uncertainty, which we will explore in section 2.3.

2.1 Heuristics

Though the execution of a real Spark query is highly complex, we are able to get relatively accurate estimates of the query execution times based on a simplified version of the Spark FIFO task scheduler. Our simplified version, however, makes use of a number of heuristics. These heuristics allow us to estimate the parallel execution of different stages (section 2.1.1), the number of tasks (section 2.1.2), the size of each task (section 2.1.3), and the run time of each task (section 2.1.4).

2.1.1 Parallel Stages. Spark's ability to run stages in parallel come from the fact that the code used to implement the SQL engine is multi-threaded. To avoid simulating the exact code of Spark's SQL engine, we create a task graph, such as figure 1. Using the task graph, we use a rule-based model to simulate the Spark stage execution.

Because of Spark's FIFO task scheduler, we can use the heuristic that every stage launches all of its tasks before any other stage can begin launching tasks. Formally, given a query Q a set of stages $S = \{s_1, \cdots, s_m\}$. Thus, stage $s_i \in S$, at any point, if stage s_i is launching new tasks, then stage s_j where $i \neq j$ cannot be launching new tasks.

Additionally, every stage has to wait for its parent stages to finish, which when all of the stage's tasks finish running. Thus, there are instances when there are nodes free, but the next stage in FIFO order is blocked by its parent. In such cases, we begin the next stage that is not blocked by its parents or wait until the parent stages have finished. Thus, given a stage s_i with a parent stage s_{i-1} and a stage s_{i+1} such that there is no path in the stage DAG from s_{i-1} to s_{i+1}, if s_{i-1} still has tasks running, then stage s_i cannot launch any tasks. However, stage s_{i+1} can run even though it is after s_i in terms of FIFO order, assuming its parent stages have completed. When s_{i+1} has completed, if s_{i-1} has yet to complete, then s_i must wait until s_{i-1} has completed before s_i can begin execution of any tasks.

2.1.2 Task Count. Although the number of tasks for each Spark stage is dependent on a number of factors, we can ignore a number of those factors due to the fact that we have the trace of a previous execution of the query. Instead we focus on the fact that there is a minimum and maximum number of tasks for a particular stage regardless of the number of nodes in the cluster. Thus, we estimate the number of tasks for a given stage by comparing the number of tasks in the trace of the previous execution, the number of nodes in the previous execution, and the number of nodes in the cluster for which we are predicting the query execution time. Thus, if the number of tasks the trace is not equal to the number of nodes used when collecting the trace, then we set the number of tasks to be the number of tasks from the trace. We also set the number of tasks to the number of nodes in the cluster when the number of nodes exceeds the number of tasks. Otherwise, we estimate the number of tasks to be equal to the number of nodes in the cluster for which we are estimating the query execution time.

2.1.3 Task Size. While there is some variability in the amount of data that each task is given, this variability is small enough such that it has a negligible effect on the run time of the query. Thus, for our model, we ignore the fact that there is any variability in the size of data executed per task. As a result, we set the amount of data equal to the median amount of data that the task is given.

In order to account for the fact that the total data size should not change when the number of tasks change, we use the heuristic that the task size is proportional to the number of tasks, in both the previous and estimated execution, to account for the change in the number of tasks. In doing so, we ensure that the total amount of data that each stage intakes is the same. Formally, each stage s_i has a set of tasks $\mathcal{T}_i = \{\tau^1, \cdots, \tau^t\}$ Thus, if the previous execution had

t_e tasks and median amount of data handled for each task $\tau^{(p,i)}$ was $\hat{\tau}_b^{(p,i)}$, then the estimated execution, with t_e tasks, has task $\tau^{(e,i)}$ handles $\tau_b^{(e,i)}$ bytes of data given by

$$\hat{\tau}_b^{(e,i)} = \frac{t_p}{t_e}\hat{\tau}_b^{(p,i)} \tag{1}$$

2.1.4 Task Run Time. To simplify our model of the task execution times, we model the task durations of the same stage from a log Gamma distribution. We use the log Gamma distribution because of its nonnegativity and long and heavy tail. Although the Gamma distribution also has these properties, the log Gamma distribution can accurately represent normally distributed data.

We account for the fact that the run times are dependent on the data size by normalizing the task run times by the task size before fitting it to a log Gamma distribution. In doing so, we ensure that we isolate the variations in run time from the amount of data. Formally, given a stage s_i with task $\tau^j \in \mathcal{T}$, each task has a duration of τ_d^j and takes in τ_b^j bytes of data. We assume that

$$\frac{t_d^i}{t_b^i} \sim LogGamma(k, \theta) \tag{2}$$

To get the run time of a task in a particular cluster provision, we simply obtain the estimated task size and multiply it by a sample from the above distribution.

We also currently do not model the task scheduling time, due to the fact that the scheduling time is negligible given that the amount of data handled by each task is reasonable. Additionally, we assume that task run times capture network latencies, so we do not specifically handle these latencies.

2.2 Simulation

Using the above assumptions, we are able to simulate the execution of a query in Spark outlined in algorithm 1.

To do so, we iterate through the stages in an order based off of the heuristics from section 2.1. For each stage, simulate the task durations (line 16 - line 22). To do so, we first estimate the number of tasks per stage, the amount of data for each stage, and we fit a log Gamma distribution to the task durations normalized by the amount of data for each task. To fit the log Gamma distribution to the data collected from the previous execution, we use maximum likelihood estimation in order to determine the parameters to the distribution. Then, we can determine the task durations by sampling the log Gamma distribution for the ratio of the task duration for each byte handled by the task. We then recover the task duration by multiplying the ratio (i.e., the sampled valued) by the number of bytes handled by the task.

With the task durations, we simulate a cluster of n_e nodes (line 23 - line 25). To do so, we create a *min-heap* of size n_e that will hold the finish times of the n_e tasks that we simulate to be running in parallel. Thus, time only advances when we have to wait for a task to complete.

Using the algorithm that we described, we are able to estimate the run time of the entire query. The estimate run time of the entire query is the time when all tasks have been completed in the simulated cluster.

Algorithm 1 Spark Simulator

```
1:  Given: Stages in query trace, S
2:  Given: Number of nodes to estimate, n_e
3:
4:  cluster ← minHeap(n_e)
5:  time ← 0
6:  i ← 0
7:  while i < m do
8:      begin
9:      if noneCanRun() then
10:         time ← cluster.max()
11:         cluster.empty()
12:     if canRun(s_i) then
13:         i ← i + 1
14:         continue
15:     S.pop(i)
16:     t̂_i ← estimateTaskCount(n_e, Q)
17:     τ̂_b^i ← estimateTaskSize(n_e, Q)
18:     model ←logGamma.MLE_fit(s_i)
19:     for j = 1 to t̂_i do
20:         begin
21:         r ← model.sample()
22:         τ̂_d ← τ̂_b * r
23:         if cluster.full then
24:             time ← cluster.popAllMin()
25:         cluster.insert(time + τ̂_d^j)
26:         end
27:     i ← min(S)
28:     end
29: time ← cluster.max()
```

2.3 Sources of Uncertainty

Due to the fact that the trace of a previous execution is a sample of the task durations, the numerous heuristics we are using, and the simulation of the task durations, we have three sources of uncertainty that need to be captured within the Spark Simulator: (1) σ_s, the *sample uncertainty*, in section 2.3.1 (2) σ_h, the *heuristic uncertainty*, in section 2.3.2, and (3) σ_e, the *estimate uncertainty*, in section 2.3.3. In order to capture these uncertainties, we let σ_s, σ_h, and σ_e be the sample task duration standard deviation, the standard deviation of the task duration using different heuristics, and the standard devation of the estimates. To get the total uncertainty, σ, we sum together the different sources of uncertainties:

$$\sigma = 3(\alpha_s\sigma_s + \alpha_h\sigma_h + \alpha_e\sigma_e) \tag{3}$$

where α_s, α_h, and α_e are used to adjust the weights of the different sources of uncertainty. Furthermore, we avoid scaling the overall uncertainty by an arbitrary factor by enforcing

$$\alpha_s + \alpha_h + \alpha_e = 1$$

Currently, we set $\alpha_s = \alpha_h = \alpha_e = \frac{1}{3}$. We discuss the sources of these uncertainties and how we calculate their standard deviations below.

2.3.1 Sample Uncertainty. The sample uncertainty comes from the fact that the task run time is not constant after normalizing for

the task data size. This comes from the fact that the run time is dependent on the values of the data and not just the size itself. Additionally, there is variability in the cluster and task execution, such as the distance between the nodes in the cluster and the effects of multi-tenancy on the nodes. As a result, there is a noninsignficiant variability in the task run times even after normalizing for the data size.

Although calculating the standard deviation of the task durations (normalized by the task size) for stage s_i from the previous execution, $\sigma_{(s,s_i)}$, is trivial, the difficulty lies in using these $\sigma_{(s,s_i)}$'s to calculate its effect on the overall query. This difficulty arises from the fact that, though stages launch all of its tasks at once, the tasks are not guaranteed to all finish at the same time. While we could approximate that they all finish at around the same time, we are unable to do so due to straggler tasks. Straggler tasks are significant due to the fact that stages cannot finish until all tasks, including straggler tasks, have finished.

However, we are able to upper bound the uncertainty, represented by the standard deviation. The upper bound of the uncertainty comes occurs if there was only one node, which causes everything to run sequentially. Thus, we use this upper bound and let σ_s be

$$\sigma_s = \sum_{i=1}^{m} \hat{t}_c^i \hat{\tau}_b^i \sigma_{(s,s_i)} \tag{4}$$

where $\hat{t}_c^i$ is the estimated number of tasks for stage i using n_e nodes and $\hat{\tau}_b^i$ is the estimated task size for stage i.

2.3.2 Heuristic Uncertainty. The heuristic uncertainty comes from the fact that we are using heuristics rather than actually determining the task count, task size, and task durations. Thus, the heuristic uncertainty is calculated using

$$\sigma_h = \sigma_{(h,c)} + \sigma_{(h,s)} + \sigma_{(h,d)} \tag{5}$$

where $\sigma_{(h,c)}$ is the uncertainty from the task count heuristics, $\sigma_{(h,s)}$ is the uncertainty from the task size heuristics, and $\sigma_{(h,d)}$ is the uncertainty from the task duration heuristics.

Task Count Uncertainty. The uncertainty from the task count heuristics comes from the fact that, even though our heuristic will scale the number of tasks with the number of nodes when the task count is equal to the node count in the previous execution, this will not always be the case due the fact that there is a minimum and maximum task count for each stage. This minimum and maximum task count is reasonable as it corresponds to the minimum and maximum degree of parallelism. Though the minimum and maximum task count is dependent on the amount of data for each task, the amount of data for the minimum and maximum task count is also dependent on a number of other factors, which we do not consider. Thus, our current approach results in an uncertainty on the task count.

To calculate the uncertainty, we calculate the query run time deviation from our current estimate. Similar to the sample uncertainty, we are unable to calculate the uncertainty for our estimate number of nodes exactly. Instead, we are forced to use the upper bound of the uncertainty, which considers the case where there is only one node and every thing runs serially. Thus, the "standard deviation" of the task count[1] is

$$\sigma_{(h,c)} = \sum_{i=1}^{m} \frac{1}{t_c^{(p,i)} - t_c^{(e,i)}} \sum_{t=t_c^{(e,i)}}^{t_c^{(p,i)}} (t_c)(\frac{t_c^{(e,i)}}{t_c}\hat{\tau}_b^i)\hat{r}_i - (t_c^{(e,i)})(\hat{\tau}_b^i)\hat{r}_i \tag{6}$$

where $t_c^{(e,i)}$ is the estimated number of tasks, $t_c^{(p,i)}$ is the number of tasks in the previous execution trace, and $\hat{r}_i$ is the largest ratio of task duration to task size for stage i in the previous execution trace.

Task Size Uncertainty. The task size heuristic results in uncertainty due to the fact that our heuristic uses the median task size while the task size varies in actual execution. Although, for stage i, the variability in the task size in the previous execution trace, $\sigma_{(h,s,\mathcal{T}_i)}$ can be easily calculated, we are still forced to estimate the upper bound uncertainty, which occurs when we run the query serially on one node.

To calculate the standard deviation of the task size, we use the following

$$\sigma_{(h,s)} = \sum_{i=1}^{m} \hat{t}_c^i \sigma_{(h,s,\mathcal{T}_i)} \hat{r}_i \tag{7}$$

Task Duration Uncertainty. We, also, have to consider the uncertainty that comes from the task durations. This uncertainty comes from the fact that the task durations do not exactly follow the log Gamma distribution, even after we normalize it by the task size.

To calculate the uncertainty of the task durations, we cannot simply use the variance of the log Gamma distribution and, since we are using a maximum likelihood estimation, we do not have an uncertainty from the fitting. Instead, for the uncertainty of stage i, we calculate the difference between a sample of size t_c^i of the fitted log Gamma distribution and the actual task execution duration values, normalized by the task size. Similar to the above uncertainties, we calculate the upper bound of the uncertainty by calculating the uncertainty if we were executing only on one node sequentially. Thus, given $\forall i \in [1, m]\ \forall j \in [1, t_c^i],\ d_{(i,j)} \sim LogGamma(k, \theta)$, the uncertainty is

$$\sigma_{(h,d)} = \sum_{i=1}^{m} \hat{t}_c^i \hat{\tau}_b^i \sum_{j=1}^{t_c^i} d_{(i,j)} - \frac{\tau_d^{(i,j)}}{\tau_b^{(i,j)}} \tag{8}$$

where $\tau_d^{(i,j)}$ is the duration of task j from stage i of the previous execution trace and $\tau_b^{(i,j)}$ is the size of task j from stage i of the previous execution trace.

2.3.3 Estimate Uncertainty. Since we are simulating the task durations based on a log Gamma distribution, we are introducing uncertainty from the simulations. Thus, we have to account for this uncertainty in our calculations.

To calculate this uncertainty, we need to run the entire task duration model more than once to calculate the standard deviation of the task durations $\sigma_{(e,s_i)}$ for stage i. Although this uncertainty is from the simulations, we still calculate the upper bound of the

[1]The standard deviation in equation 6 is when the estimated task count is less than the task count in the previous execution trace. If the estimated task count is greater than the task count in the previous execution trace, then we switch the estimated task count with the task count in the previous execution trace and vice-versa.

uncertainty by calculating the uncertainty if we were executing only on one node sequentially. Thus,

$$\sigma_s = \sum_{i=1}^{m} \hat{t}_c^i \hat{\tau}_b^i \sigma_{(e,s_i)} \tag{9}$$

Unlike the other uncertainties, we are able to control the size of the estimate uncertainty through repeating the Spark Simulator. However, to ensure that the simulation time does not exceed the actual query execution time while simultaneously ensuring that the estimate uncertainty is small enough, we run the Spark Simulator 10 times for each cluster configuration. In doing so, we find that the estimate uncertainty is small relative to the other uncertainties and that the Spark Simulation time is still insignificant relative to the query execution time.

3 SERVERLESS SPARK SIMULATOR

In section 2, we estimated the run time of Spark queries under fixed sized cluster provisions. Although estimating the run times is necessary, it has already been done before [7, 16] In this section, we focus on how we can use query and stage run times of the Spark Simulator from section 2 to (i) determine the run times of the query in a serverless setting in section 3.1, (ii) estimate out the time-cost trade off curve in section 3.1, (iii) automatically determine a cluster provisioning given a budget in section 3.1, and (iv) automatically determine the next fixed cluster provisioning to run to improve the error bounds of the time-cost trade off curve in section 3.2. We note that some of the methods we describe below is what we are planning to do and, as a result, have no results as they have yet to be implemented.

3.1 Offline Simulator

For the offline simulator, there are two possible outputs: (i) a time-cost trade off curve that gives the time-cost trade off of the model with accompanying cluster configurations at every point in section 3.1.1 and (ii) a cluster configuration with the optimized wall-clock time or cost given a cost or wall-clock time budget, respectively, in section 3.1.2. Although the second output is trivial given a time-cost trade off curve, we can reduce the amount of computation necessary to produce one optimized cluster configuration in comparison to finding the entire time-cost trade off space.

3.1.1 Time-Cost Trade Off Curve. To calculate the time-cost trade off curve, we have to determine (i) which groups of stages that are executed in parallel, (ii) the fixed cluster configuration query execution times, and (iii) the dynamic cluster configuration query execution times. Computing these values, with the help of heuristics, allows us to compute the relevant parts of the time-cost trade off curve. Below we only discuss how we calculate the run times as the cost of the query is the product of the run time, the number of nodes used, and the cost of each node.

Parallel Stages. It is necessary to determine which stages can be executed in parallel because we currently do not make use of the fact that we can have multiple Spark clusters. Currently, we use the heuristic that if we have a large enough Spark cluster for each group of stages that can be run in parallel, then we can run all of the tasks of those parallel stages in parallel. Thus, we can get most of the benefits of multiple drivers without actually using multiple drivers. We discuss more about the impacts of this in section 6.

To determine which stages can be executed in parallel, given a large enough cluster size, we go through the stage execution graph, such as figure 1, and determine which stages have to wait for another stage to finish before it can begin execution. Thus, these stages represent the beginning of a new group of parallel stages. Formally, there is a set of sets $\mathcal{G}$ such that $\forall g_i \in \mathcal{G},\ g_i = \{s_1, \cdots, s_{l_i}\}$, where l_i is the number of stages in g_i and $\forall s_j \in g_i, s_j$ can run given all stages in g_k have completed, if $j > k$ and no stage in g_k can run, if $j < k$.

Fixed Cluster Configurations. First, we need to determine our limits on the number of nodes that we can and should use. This is important due to the fact that, under the serverless computation regime, we can, in theory, use millions of nodes at any given point. However, it is infeasible to compute the time cost trade off curve for anywhere from one to millions of nodes.

Thus, at least for the fixed cluster configurations, we only consider the number of nodes in the range of where the entire data set can fit in the cumulative memory of the nodes, n_{min}, to $n_{max} = 10n_{min}$. If during the dynamic cluster configuration we need larger clusters, then we increase n_{max} as needed. However, we will never decrease the minimum number of nodes to below n_{min} to avoid the penalty of swapping to disk.

Rather than calculating all possible fixed cluster configurations from n_{min} to n_{max}, we instead calculate the cluster configurations of $\mathcal{N} = kn_{min} \forall k \in [1, 10]$. Thus, the number of cluster configurations that we have to calculate is constant rather than linear with the number of n_{min}. Then, we simply use the Spark Simulator to get the query run times for fixed cluster sizes $n_e \in \mathcal{N}$.

Dynamic Cluster Configuration. Rather than calculating the run time of the entire query under a dynamic cluster configuration, we calculate the run time of the groups of stages that can be run in parallel, $\mathcal{G}$. Thus, for each group $g_i \in \mathcal{G}$, we first determine the maximum degree of parallelism for g_i by calculating the total number of tasks in g_i as such

$$m_t^i = \sum_{j=1}^{l_i} t_c^i \tag{10}$$

Then, we calculate the run times of g_i from n_{min} to m_t^i in steps of n_{min} (i.e., we calculate the run time of g_i with $n_{min}, 2n_{min}, \cdots, m_t^i$). For all the estimates with node count less than n_{max}, we can use the results from the fixed cluster calculations. For estimates with node count greater than n_{max} (i.e., $n_e > n_{max}$), we calculate the fixed cluster configuration run time of n_e for the entire query as well as the run time for g_i.

After computing the run times of all g_i, we can simply do a combinatorial addition starting with the mid-sized cluster configurations. We begin from the middle and expand out so that, once we reach a time or cost greater than the fixed cluster configuration value, we can stop searching.

3.1.2 Optimized Given Budget. To optimize for a given budget, we can formulate it as a dynamic programming problem. However, before we can do so, we need to calculate the fixed cluster run times. We do so using the same way that we did so in section 3.1.1.

Algorithm 2 Minimize cost given time budget

Given: Matrix of costs of running the query on n different fixed cluster configurations grouped by m groups of stages that can be executed in parallel, $C \in \mathcal{R}^{n \times m}$
Given: Matrix of run times of running the query on n different fixed cluster configurations grouped by m groups of stages that can be executed in parallel, $T \in \mathcal{R}^{n \times m}$
Given: Maximum amount of time to execute query, $t_{max} > 0$

$cost[n+1][m+1] \leftarrow 0$
$time[n+1][m+1] \leftarrow 0$
for $i = 1$ **to** $n+1$ **do**
  begin
  for $j = 1$ **to** $m+1$ **do**
    begin
    if $cost[i-1][j] + C[i-1][j] >= cost[i][j-1] + C[i][j-1]$ **then**
      if $time[i][j-1] + T[i][j-1] > t_{max}$ **then**
        $cost[i][j] \leftarrow \infty$
      else
        $cost[i][j] = cost[i][j-1] + C[i][j-1]$
        $time[i][j] \leftarrow time[i][j-1] + T[i][j-1]$
    if $cost[i-1][j] + C[i-1][j] < cost[i][j-1] + C[i][j-1]$ **then**
      if $time[i-1][j] + T[i-1][j] > t_{max}$ **then**
        $cost[i][j] \leftarrow \infty$
      else
        $cost[i][j] = cost[i-1][j] + C[i-1][j]$
        $time[i][j] \leftarrow time[i-1][j] + T[i-1][j]$
    end
  end

With the fixed cluster query run times, we break up the run times into the run times of the groups of parallel stages.

Then, we can solve for the minimum run time or cost, given a cost or run time budget, respectively, using dynamic programming, since this can be reduced to the Knapsack problem[2]. Without loss of generality, we formulate the rest of the problem if we were given a run time budget and wanted to minimize cost. We can extend this solution to if we were given a cost budget and wanted to minimize run time by switching run time with cost and vice-versa.

We begin by setting up a matrix of run times and a matrix of costs with the columns representing different groups of parallel stages and the rows representing different number of nodes in the fixed cluster. Then, we iterate through the two matrices simultaneously. We search for the minimum cost path through the cost matrix while ensuring that our cost through the run time matrix does not exceed our run time budget. If there is no path through the run time matrix that is below our run time budget, then we return that it is infeasible to execute the query with the given run time. Otherwise, we can then return the final cost and the number of nodes for each group of parallel stages by returning the path that we took through the matrices.

[2]The proof is trivial and is left as an exercise to the reader.

3.2 Sampling

Because we also report an error bound with both the time-cost trade off curve and the optimized cluster configuration given a budget, we introduce the idea of sampling to reduce the error bound to an amount acceptable to the user. We are able to reduce the error bound to be arbitrarily small due to the fact that we can always collect more data to reduce the sample and heuristic uncertainties from section 2.3.1 and section 2.3.2, respectively, similar to how we can reduce the estimate uncertainty through repeating the simulation on the same fixed cluster configuration (section 2.3.3).

Since both outputs create a number of fixed cluster configurations, each with error bounds, we formulate the problem of how to select fixed cluster configurations to run to obtain a better sample as the multi-armed bandit problem. We let each of the estimated run times of the fixed cluster configurations be an "arm" in the problem and the heuristic uncertainty be the value that we try to minimize. We solve the multi-armed bandit problem by looking for the largest heuristic uncertainty.

4 RESULTS

To demonstrate the results of our work so far, we show two sets of results: the ideal results that we can achieve in section 4.1 and the simulation results in section 4.2.

4.1 Ideal Results

To demonstrate the performance gains achievable through serverless computation, we run a Spark script that executes common data science queries from a Spark tutorial [1] on the NASA HTTP server data [6]. Although the regular NASA HTTP data is only 200 MB, we replicated it 25× to reach a size of 5 GB and stored it in AWS S3. The executed the script on AWS EC2 m5.large (2 CPU and 4 GB RAM) instances with 2 to 64 nodes. Though the cost of a m5.large node is $0.09 per hour, we use a cost of $1 per second for ease of comprehension.

4.1.1 Performance Gains. We investigated the possible improvements of our system through two different approaches. First, we determined the possible improvements that would arise from improving the fact that we were able to parallelize stages of the task graph that were able to be parallelized. Then, we investigated the benefits of dynamically changing the cluster size. We present these results in table 2. For these experiments, we assume that network bandwidth would not be a bottleneck and that there would be enough memory to store the intermediate results on the reaminder of nodes.

Parallelized Stages. To investigate the effects of parallelized stages, We simply replicating the cluster configuration to each driver. For instance, if the fixed cluster provisioning had three nodes, we would parallelize stages by giving each driver three nodes.

From table 2a, it is clear that we can get performance gains of between 35% to nearly 50% from simply parallelizing stages that can be parallelized. Additionally, there is only a 0.1% to 5% cost overhead that results from the parallelism.

These results are reasonable because the compute time does not change, as we are still using the same algorithms regardless of the parallelism. However, we are able to improve the parallelism by

Value	2 Nodes	4 Nodes	6 Nodes	7 Nodes	8 Nodes	12 Nodes	16 Nodes	32 Nodes	64 Nodes
Fixed Cluster Time (s)	1,480	681	443	381	350	236	192	120	75
Fixed Cluster Cost	$2,960	$2,726	$2,661	$2,670	$2,802	$2,837	$3,082	$4,168	$4,800
Naive Serverless Time (s)	767	363	256	213	191	134	109	76	47
Naive Serverless Cost	$2,968	$2,742	$2,685	$2,698	$2,834	$2,885	$3,146	$4,296	$5,056
Naive Time Improvement	48%	46%	42%	44%	45%	43%	43%	41%	36%
Naive Cost Improvement	-0.2%	-0.6%	-0.9%	-1%	-1%	-2%	-2%	-3%	-5%

(a) Table shows the time and cost of fixed sized clusters and naively parallelizing the cluster in a serverless setting. It is clear that there are a lot of performance gains from simply parallelizing the cluster into a serverless setting.

Value	2 Nodes	8 Nodes	64 Nodes
Fixed Cluster Wall-Clock Time (s)	1,480	350	75
Fixed Cluster CPU Time (s)	2,960	2,802	4,800
Fixed Serverless Wall-Clock Time (s)	767	191	47
Fixed Serverless CPU Time (s)	2,968	2,834	5,056
Fixed Wall-Clock Time Improvement	48%	45%	36%
Fixed CPU Time Improvement	-0.2%	-1%	-5%

Value	2 Nodes	8 Nodes
Fixed Cluster Wall-Clock Time (s)	1,480	350
Fixed Cluster CPU Time (s)	2,960	2,802
Fixed Serverless Wall-Clock Time (s)	767	191
Fixed Serverless CPU Time (s)	2,968	2,834

Value	Serverless 8 & 12 Nodes	Serverless 8, 64, & 12 Nodes	Optimized Serverless
Single Driver Time (s)	332	251	997
Single Driver Cost	$2,697	$3,531	$2,248
Multi-Driver Time (s)	190	136	593
Multi-Driver Cost	$2,734	$3,568	$2,295
Multi-Driver Time Improvement	42%	45%	40%
Multi-Driver Cost Improvement	-1%	-1%	-2%

(b) Table shows the performance gains of single and multi-driver clusters in a serverless setting. It is clear that, by further adjusting the sizes of the cluster, we can push and expand the Pareto curve beyond the fixed cluster provisioning Pareto curve.

Value	Optimized Serverless
Dynamic Cluster Wall-Clock Time (s)	997
Dynamic Cluster CPU Time (s)	2,248
Dynamic Serverless Wall-Clock Time (s)	593
Dynamic Serverless CPU Time (s)	2,295
Dynamic Serverless Wall-Clock Time Improvement	40%
Dynamic Serverless CPU Time Improvement	-2%

(c) Table shows the performance gains of single and multi-driver clusters in a serverless setting. It is clear that, by further adjusting the sizes of the cluster, we can push and expand the Pareto curve beyond the fixed cluster provisioning Pareto curve.

Table 2: These tables show the performance gains that are possible through better parallelism and improved server configurations.

using more nodes. It is noteworthy that, as the number of nodes increase, the time improvement decreases and the cost penalties increase. This is the case because we already are reaching the maximum degrees of parallelism, as the number of nodes increase. Additionally, the overhead for parallelism increases since we have to pay the overhead for more nodes.

Dynamically Sized. To investigate the dynamic sizing of clusters, we simulated the changing of cluster sizes over a simulated network with a 10 Gbit/s network.

For the first two serverless columns in table 2c, we manually looked for node counts that would give improvements over fixed sized clusters. For the first column of table 2c, we found that changing the number of nodes associated with each driver from 8 to 12 nodes in the middle of the query would result in a 12% time improvement over only using 7 nodes while only causing the cost to increase by 1%. However, if we used multiple drivers by replicating the cluster to each driver, we get an improvement of 50% in run time and only a 2% increase in cost compared to only using 7 nodes.

On the other hand, if we changed the number of nodes associated with each driver from 8 to 64 and then to 12, we get a cost savings of 15% but the run time increases by over 90% compared to 32 nodes, which is more expensive and slower than 16 nodes. However, if we used multiple drivers, we see an improvement of over 50% in run time while getting a 15% increase in cost compared to 16 nodes.

These improvements show that a majority of the improvements come from the fact that we are able to leverage multiple drivers. However, there are also significant improvements to be made by ensuring that ever stage has its optimal number of nodes. Although the performance improvements of dynamically changing clusters are small relative to the performance improvements that result from naively launching multiple drivers, there is still a lot of room for improvement as these results are from manually determining which cluster sizes are best for each node.

4.1.2 Budget Optimization. Using the results that we obtained in table 2a, we optimized a task given a run time budget of 1000 seconds with the results in table 2c. We can see that the dynamic programming algorithm that we proposed in section 3.1.2 found a cluster configuration in under 1 second for each stage such that the cost was over 10% cheaper than any fixed cluster cost while meeting our run time budget.

Additionally, the result that we found was under the condition that we were only using one driver. However, even when we manually generalize this to multiple drivers, we find that, though the run time is sill greater than any of the fixed cluster size run times, we are able to keep the cost over 10% cheaper than any of the configurations in table 2.

This result is significant because it shows that by simply adjusting the cluster size to meet each the ideal parallelism, we are able to push the minimum cost beyond the minimum of any fixed sized cluster. However, this does come at the expense of taking over 2× slower than any of the fixed cluster offerings. Although the run time is increased, it should not overshadow the fact that we are able to further decrease the cost of running queries.

4.2 Simulation Results

To test our simulations, we use the TPC-DS's query 9 with a scale factor of 20 on 4, 8, 16, 32, and 64 AWS EC2 m5n.large (2 CPU and 4GB RAM) nodes. Using these execution traces, we ran the Spark Simulator from section 2. Each of these simulations took approximately 7 seconds on a modern laptop (4 CPUs). However, we can reduce the run time of the simulations by using a machine with more nodes for increased parallelism.

It is clear from figure 2 that using the upper bound of the uncertainty discussed in section 2.3 has ensured that the actual run time is included within the error bounds. However, the error bounds are also so big such that they are no longer useful.

Furthermore, when the trace has more nodes than regular, we see, in figure 2a and figure 2b for 64 and 32 nodes respectively, the Spark Simulations drastically underestimate the duration. This is the case due to the fact that the Spark Simulator assumes that the number of tasks scales with the number of nodes if they are equal in the trace. However, they reach a minimum and cause the run time to increase when the number of nodes decrease. Although we do account for this underestimate of the run time in the uncertainty, we overestimate the uncertainty when actually calculating the uncertainty caused by the task heuristics in equation 6.

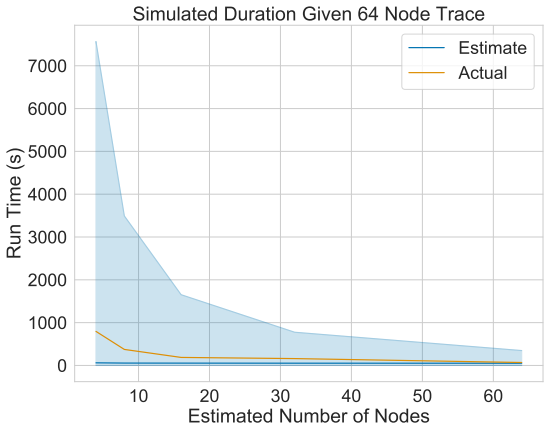
However, when we use a smaller number of nodes in our previous execution trace, we notice that, in figures 2c and figure 2d for 16 and 8 nodes respectively, the estimate closely follows the actual execution run times. However, there is a relatively larger deviation at 32 nodes. This deviation is caused by two factors: (i) the task run time normalized by the task size changes with the number of nodes and (ii) the simulator is better able to estimate clusters with a large number of nodes. The first factor, the deviation of the task run time normalized by the task size, can be explained by the fact that Spark breaks up a query into stages. These stages are divided whenever Spark is forced to do a shuffle. Thus, for each task, there is some form of a shuffle. However, as we increase the number of nodes, there is a point where the shuffle overhead is no longer trivial relative to the performance gains from increased parallelism, leading to a longer task run time normalized by task size.

The second factor, the simulator is better able to estimate clusters with a large number of nodes, is due to the fact that the first factor is offset by the increased number of tasks. The increase in the number of tasks increases the likelihood of drawing a sample from the log Gamma distribution that is from the tail, which represents a straggler task.

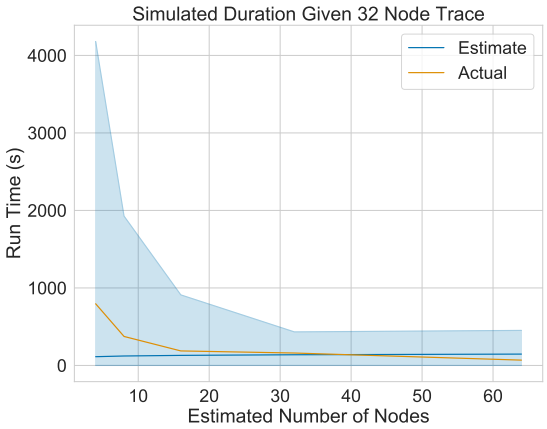
Thus, as long as our initial execution of the query to use as the trace has a small number of nodes, we are able to get relatively accurate estimates.

5 RELATED WORK

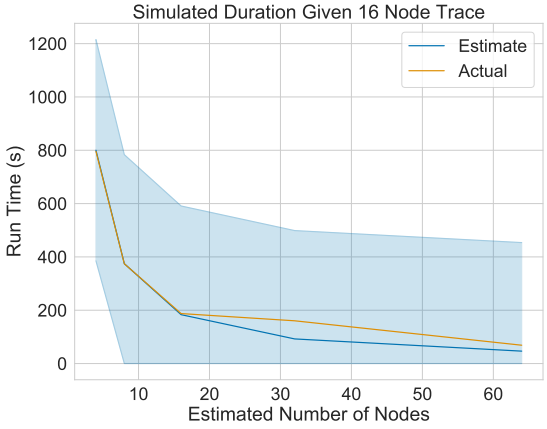
Selecting cloud configurations for computation. There already exist systems that can pick the optimal cluster configuration [7, 13, 16]. These systems, however, determine the optimal cluster configuration for a serverful computation setting. Furthermore, these systems do not attempt to return the entire time-cost trade off curve. As a result, these systems have a much more simpler model that is based on linear regression [16] and Bayesian optimization [7].



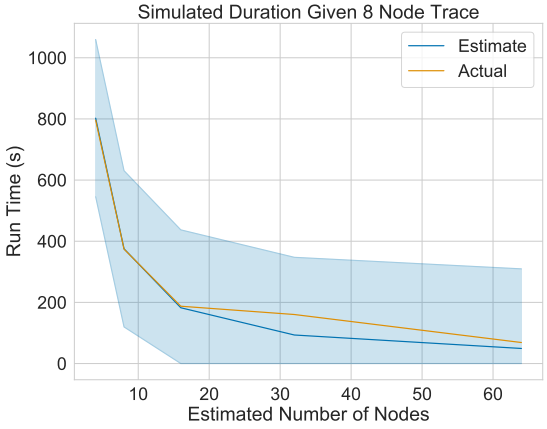
(a) Simulated results given a trace from a 64 node cluster.



(b) Simulated results given a trace from a 32 node cluster.



(c) Simulated results given a trace from a 16 node cluster.



(d) Simulated results given a trace from a 8 node cluster.

Figure 2: The plots above show the simulation results with error bounds of one standard deviation. Although we can estimate the results within the error bounds, the error bound is too big to be useful.

However, there are systems [13] that profile the query execution environment, similar to our Spark Simulator. Theses systems profile the query execution environment in a finer granularity rather than creating models simulating the execution envronment, such as our Spark Simulator.

Selecting cloud configurations for storage. On the other hand, there are also systems that can dynamically adapt cluster sizes to optimally meet service-level agreements, such as cost or latency, These systems include key-value stores, such as Anna [17] that move the data across different the storage systems (e.g., AWS EBS from AWS S3) to optimize for cost or latency. However, these systems are designed primarily for data warehousing rather than query execution. There are also other systems [9, 10] that optimize for both the data storage and query execution. These systems build only the data storage layers on the ideas of serverless computation but the query execution are built on serverful computation environments.

Optimizing query execution costs. Additionally, there are other systems [11, 14, 18] that focus on the cost of executing queries. These systems are built to ensure that they minimize the cost of executing queries either in a database or a mechanical Turk setting. However, these systems focus in a trade off space other than the cluster provisioning. They focus on exploring the tradeoffs between using a parallel database or map-reduce systems [14], trying to optimally price their jobs to ensure on-time execution in transient systems [18], or dynamically adjust the price for crowdsourced queries to ensure on-time completion [11].

Serverless execution. Finally, there has been exploration of building systems for serverless computation. This developing in the serverless space lead to the development of new serverless platforms [8**?**] or analysis of current serverless platforms [12]. These systems, however, have focused on either building a map-reduce paradigm

within a serverless context [15] and improving the communication overheads that are associated with serverless computation [8].

6 FUTURE WORK

For future work, we plan to improve both the Spark Simulator and the Serverless Spark Simulator as there is still a lot of room for improvement on top of the current work presented. This future work will improve the current system and lead to more performance gains.

6.1 Spark Simulator Improvements

Overall the Spark Simulator is relatively accurate based on the results from section 4.2. However, there is still room for improvement in terms of the task heuristics, the calculation of the uncertainty, and general improvements.

6.1.1 Task Heuristics Improvements. For the task heuristics, we plan to improve the task count estimation and the task run time, normalized by task size, estimation. The task count estimation currently assumes that the task count will scale with the number of nodes if they were equal in the trace of the previous execution. However, from section 4.2, we have seen that this can lead to a significant deviation and forces us to place a large uncertainty on the overall query execution time. Through more investigation into the Spark query planner, we should be able to more accurately estimate the number of tasks, which leads to a more accurate estimate and smaller error bounds.

Additionally, the estimation of the task run time can also benefit from switching over to a Bayesian approach towards fitting the log Gamma distribution. Although the maximum likelihood estimation that we are currently using is sufficient, a Bayesian approach towards fitting will allow us to model stages with only one task and easily combine the data from multiple traces. Using a Bayesian approach, with a reasonable prior, will allow us to model stages on the log Gamma distribution even if there is only one data point. Additionally, by using a prior, we can increase the accuracy of the model by only adding in the new data without having to fit based on all the data we have collected so far. We also need to better understand how the task run times, normalized by task size, changes when the number of nodes change.

6.1.2 Uncertainty Calculation Improvements. From section 4.2, the actual run times do fall within one standard deviation of our estimated run times, but our calculation of the standard deviation is too big to be helpful. Thus, we plan to improve our uncertainty calculations such that we can apply some simple heuristics to avoid having to use the upper bound of the uncertainty.

6.1.3 General Spark Simulator Improvements. The most important line of work that we plan to continue with is being able to estimate the run time of the query on the entire data set given a trace of the previous execution on a sample of the data set. Although we already have the framework in place to handle such an estimate, Spark's query planning changes significantly when the amount of data it handles changes. Thus, there is still a lot of work to be done in understanding Spark's internals before we can integrate such a feature.

6.2 Spark Serverless Improvements

From section 4.1, we have seen that using multiple drivers can lead to significant improvements. However, our current approach towards simulating a serverless environment only estimates using one driver. Thus, extending our current implementation of the Serverless Spark simulator should lead to significant improvements.

7 CONCLUSION

We have shown that serverless offerings have the potential of being 2× faster than serverful offerings at approximately the same price. Furthermore, we are developing a model that takes in an arbitrary query and sampled data to return a time-cost tradeoff profile with corresponding cluster provisioning. In doing so, it will provide users an understanding of how their queries will perform at various price points while simultaneously providing service providers with the dynamic cluster provisioning for the user's desired performance.

REFERENCES

[1] 2019. Apache Spark Tutorial: Getting Started with Apache Spark on Databricks. https://databricks.com/spark/getting-started-with-apache-spark

[2] 2019. AWS Athena. https://aws.amazon.com/athena/

[3] 2019. AWS Lambda. https://aws.amazon.com/lambda/

[4] 2019. GCP BigQuery. https://cloud.google.com/bigquery/

[5] 2019. GCP Function. https://cloud.google.com/functions/

[6] 2019. NASA-HTTP. ftp://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

[7] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. 469–482. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[8] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing - SoCC '19*. ACM Press, Santa Cruz, CA, USA, 13–24. https://doi.org/10.1145/3357223.3362711

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems* 26, 2 (June 2008), 1–26. https://doi.org/10.1145/1365815.1365816

[10] Carlo Curino, Evan Jones, Raluca Popa, Nirmesh Malviya, Eugene Wu, Samuel Madden, Hari Balakrishnan, and Nickolai Zeldovich. 2011. Relational Cloud: A Database-as-a-Service for the Cloud. In *CIDR 2011 - 5th Biennial Conference on Innovative Data Systems Research, Conference Proceedings*. 235–240.

[11] Yihan Gao and Aditya Parameswaran. 2014. Finish them!: pricing algorithms for human computation. *Proceedings of the VLDB Endowment* 7, 14 (Oct. 2014), 1965–1976. https://doi.org/10.14778/2733085.2733101

[12] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. *arXiv:1812.03651 [cs]* (Dec. 2018). http://arxiv.org/abs/1812.03651 arXiv: 1812.03651.

[13] Herodotos Herodotou, Fei Dong, and Shivnath Babu. 2011. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*. ACM Press, Cascais, Portugal, 1–14. https://doi.org/10.1145/2038916.2038934

[14] Adrian Daniel Popescu, Debabrata Dash, Verena Kantere, and Anastasia Ailamaki. 2010. Adaptive query execution for data management in the cloud. In *Proceedings of the second international workshop on Cloud data management - CloudDB '10*. ACM Press, Toronto, ON, Canada, 17. https://doi.org/10.1145/1871929.1871933

[15] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*. 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[16] Shivaram Venkataraman, Zongheng Yang, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. 363–378. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman

[17] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2019. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment* 12, 6 (Feb. 2019), 624–638. https://doi.org/10.14778/3311880.3311881

[18] Fei Xu, Haoyue Zheng, Huan Jiang, Wujie Shao, Haikun Liu, and Zhi Zhou. 2019. Cost-Effective Cloud Server Provisioning for Predictable Performance of Big Data Analytics. *IEEE Transactions on Parallel and Distributed Systems* 30, 5 (May 2019), 1036–1051. https://doi.org/10.1109/TPDS.2018.2873397

[19] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010.* https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets