SplitServe: Efficiently Splitting Apache Spark Jobs Across FaaS and IaaS

Aman Jain
Pennsylvania State University
axj182@psu.edu

Ata F. Baarzi
Pennsylvania State University
azf82@psu.edu

George Kesidis
Pennsylvania State University
gik2@psu.edu

Bhuvan Urgaonkar
Pennsylvania State University
bhuvan@cse.psu.edu

Nader Alfares
Pennsylvania State University
nna5040@psu.edu

Mahmut Kandemir
Pennsylvania State University
kandemir@cse.psu.edu

CCS Concepts • **Computer systems organization** **Cloud computing**; • **Information systems** **Cloud based storage**; *MapReduce-based systems*.

Keywords Cloud Computing; Cloud Functions; Virtual Machines; Apache Spark

ACM Reference Format:
Aman Jain, Ata F. Baarzi, George Kesidis, Bhuvan Urgaonkar, Nader Alfares, and Mahmut Kandemir. 2020. SplitServe: Efficiently Splitting Apache Spark Jobs Across FaaS and IaaS . In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3423211.3425695

Abstract — Due to their lower startup latencies and finer-grain pricing than virtual machines (VMs), Amazon Lambdas and other cloud functions (CFs) have been identified as ideal candidates for handling unexpected spikes in simple, stateless workloads. However, it is not immediately clear if CFs would be similarly effective in autoscaling complex workloads involving significant state transfer across distributed application components. We have found that, through careful design, currently available CFs can indeed be useful even for complex workloads. To demonstrate this, we design and implement SplitServe, an enhancement of Apache Spark. If not enough executors on existing VMs are available for a newly arriving latency-sensitive job, SplitServe is able to use CFs to quickly bridge this shortfall in VMs, so avoiding the startup latencies of newly requested VMs. If desirable in terms of performance or cost, when newly requested VMs, or executors on existing VMs, do become available, SplitServe is able to move ongoing work from CFs to them. Our experimental evaluation of SplitServe using four different workloads (either on a mixture of VM-based executors and CFs or just CFs) shows that it improves execution time by up to (a) 55% for workloads with small to modest amount of shuffling, and (b) 31% in workloads with large amounts of shuffling, when compared to only VM-based autoscaling.

Middleware '20, December 7–11, 2020, Delft, Netherlands

ACM ISBN 978-1-4503-8153-6/20/12…$15.00
https://doi.org/10.1145/3423211.3425695

1 Introduction

Many customers (tenants) are motivated to migrate to the public cloud because of the significant savings in infrastructure costs for their private clouds. Having migrated, tenants often realize that they can further reduce costs, subject to performance requirements, by more careful management of their workloads and of the resources they procure. In particular, they can take advantage of the diversity in available cloud services, including consideration of how resources are provisioned for them and how they are billed in concert with more intelligent characterization of the resource needs of their workloads.

Public cloud providers now offer a plethora of service types ranging from infrastructure as a service (IaaS), most prominently virtual machine (VM) instances, software as a service (SaaS), and the freshly trending "serverless computing" or platform as a service (PaaS). Perhaps the most popular form of PaaS is Function-as-a-Service (FaaS), or cloud functions (CFs), that allows a tenant to execute custom functions within lightweight containers or virtual machines (VMs)[1] managed by the cloud. Commercially available examples of CFs include offerings from Amazon (AWS Lambda) [3], Google (Google Cloud Functions) [21], Microsoft (Azure Functions) [6], and IBM (IBM Cloud functions) [23][2].

CFs offer much lower startup latencies than general-purpose VMs, about 100ms when warm for AWS Lambdas vs. 2 minutes or more for AWS VMs ("instances"), respectively, with a lower minimum price. This has made it appealing to use them for handling unexpected spikes in mostly stateless workloads

[1] AWS has used a specialized small footprint VM called Firecracker instead of containers for its Lambdas since 2018 [1, 18].
[2] In the rest of this paper, we will use Lambda to refer to the AWS offering and FaaS or CF when discussing this type of offering more generally.

that are latency-sensitive [32, 37, 49]. Upon a surge in workload beyond what the provisioned VM capacity can handle, CFs can be launched quickly and excess workload can be directed their way. If the spike persists, additional VMs may be launched and CFs can be decommissioned when the VMs are ready. This is desirable both because: (a) VMs are cheaper over long periods due to their lower per-unit resource price, and (b) current CFs often have a limited lifetime (*e.g.,* 15 minutes for Lambdas).

It is not immediately clear if using CFs for latency-critical workloads that exchange significant state among distributed application components can be similarly effective. Reasons to be skeptical include the following limitations in currently offered CFs: (i) relatively limited resource capacity (*e.g.,* Lambdas may only have at most 3GB main memory, as of this writing), (ii) limited lifetime, and (iii) limited support for sharing the intermediate state (*e.g.,* Lambdas must use an external storage service such as AWS S3). Contrary to what conventional wisdom might suggest, we show that it is indeed possible to exploit the faster startup times of CFs to improve the cost/performance of autoscaling even for complex workloads.

Approach and Contributions: Our general approach consists of **(a)** launching Lambdas to handle workload in excess of the currently provisioned VM capacity and **(b)** later, if needed, segueing from these CFs to newly available VMs (or to executors that become available on existing VMs). For complex workloads, however, a number of design choices must be made to address the limitations of CFs (*e.g.,* [24, 25, 35, 44]) as well as idiosyncrasies of the application. We demonstrate how all these challenges may be addressed by embodying our approach in SplitServe, an enhancement of Apache Spark [39, 48], that allows specified tasks comprising a parallel Spark job to run on AWS Lambdas with the rest simultaneously running on VMs (the latter being the default). To accomplish this, we make three important enhancements to Spark:

- We enhance the Spark master[3] so it may launch executors on both VMs and Lambdas and divide a *single* job's tasks across them.
- We implement a high-throughput storage layer for intermediate data shuffling using HDFS, which is suitable for both VM and Lambda based executors.
- We implement mechanisms for terminating executors running on Lambdas to efficiently segue to VM-based executors without triggering extensive execution rollback due to Spark's fault recovery.

We evaluate SplitServe using diverse benchmark workloads with different data shuffling intensities. Recall that our interest is in **latency-critical** workloads wherein the user expects a response within a short amount of time. Therefore, we employ DataBrick's Spark-SQL-Perf [40] TPC-DS benchmark which utilizes a generalized query model. This model allows the benchmark to capture important aspects of the interactive, iterative nature of on-line analytical processing (OLAP) queries many of which are constructed to answer immediate and specific business questions [31, 34]. Additionally, we use Intel's HiBench [22] machine learning (K-means clustering) and web-search related (PageRank) workloads. Although these ML and search workloads are generally considered to be of the "batch processing" type where the metric of interest is throughput as opposed to latency, they still offer useful evaluation scenarios. There are in fact scenarios where latency matters even for such workloads, for e.g., when using ML or big data computing in finance [42]. Our evaluation results reveal that SplitServe is suitable for fast autoscaling using AWS Lambdas and provides advantages over purely VM- or Lambda-based execution. Specifically, by *jointly* using VMs and Lambdas to execute Spark jobs, we find up to 55% execution time improvement for jobs with small to modest amount of shuffling and up to 31% improvement jobs with large amounts of shuffling when compared to only VM-based autoscaling. SplitServe code and all our experimental data is open-sourced and freely available online [41].

[3]Note that the Spark master must itself be on a VM since it is a long-running entity.

2 Related Work

Aspect ⟶	Uses VMs?	Uses CFs?	Execution time	Cost
TR-Spark [46]	Yes	No	No	n/a
Apache Flink [8]	Yes	No	Yes	Yes
Burscale [7]	Yes	No	Yes	Yes
Qubole [36]	No	Yes	No	No
Flint [26]	No	Yes	No	No
ExCamera [20]	No	Yes	n/a	n/a
numpywren [38]	No	Yes	No	No
PyWren [24]	No	Yes	No	No
Locus (PyWren+Redis) [35]	No	Yes	Yes	No
Cirrus [25]	No	Yes	Yes	No
gg [19]	No	Yes	Yes	No
FEAT [32], MArk [49]	Yes	Yes	n/a	n/a
SplitServe	Yes	Yes	Yes	Yes

Table 1. A comparison of SplitServe against the state-of-the-art platforms exploiting VMs and Cloud Functions (CFs). The rightmost two columns relate to whether SplitServe shuffling compares favorably in terms of execution-time performance and cost to Vanilla Spark running on public-cloud VMs.

Table 1 summarizes the most important ways in which our proposed solution differs from the state of the art solutions, most which making a case for either IaaS or FaaS but not a combination of the two. Application frameworks like Apache Spark [39], and Apache Flink [8], a framework and distributed processing engine for stateful computations over unbounded and bounded data streams, employ only VMs to

run, complex Map-Reduce and bulk synchronous processing (BSP) workloads [14, 28]. These workloads typically generate a large amount of intermediate shuffle data that are either communicated directly between functions, or are persisted in storage. Even though these frameworks support dynamic executor allocation, scaling down the cluster has always been a tedious problem [12]. This is largely because, even after flushing their task queues, executors may continue their role as servers for the shuffle data for other workers. Given the fault tolerance mechanisms built into these frameworks, there may be cost overheads and performance degradation due to the transferal of large amounts of state-data for execution rollback. Furthermore, if an executor is lost, the entire execution dependency is rebuilt using the transformation lineage that Spark internally maintains.

TR-Spark [46], runs as a secondary background task on transient resources and curbs performance degradation with fleeting executors using periodic checkpointing of the its resilient distributed datasets (RDDs) [47]. I/O operations are fast and cost-effective since shuffle writes are to the local storage of the worker/slave nodes, instead of some external storage. However, since the executors are usually large in size, the straggler problems common to BSP workloads remain.

The tenant can optionally employ burstable VMs *e.g.,* [43]. A recent system called BurScale makes a case for provisioning burstable VMs as "standby" resources to deal with transient overloads while new regular VMs are being procured from the cloud by an autoscaler [7] thereby helping hide performance degradation during the possibly slow addition of new VMs. One benefit of BurScale over SplitServe is that it does not require modifications to the applications considered whereas SplitServe requires non-trivial enhancements to Apache Spark. On the other hand, BurScale deals with relatively simple applications (replicarted web servers and Memcached-based caching) and its efficacy for complex applications like Spark remains to be demonstrated. Also, BurScale's efficacy relies on being able to manage token state properly despite workload uncertainty, a complexity SplitServe does not face. Generally, we view these as solutions with complementary strengths and certain tenants might in fact find it beneficial to combine these ideas into a single system. Even if a BurScale like system for Spark were devised, the motivation for SplitServe remains: it's possible that a latency critical job will arrive to find insufficient executors (or credits) available among the VMs and burstable VMs have the same slow spin-up delay of regular VMs, *i.e.,* autoscaling with burstables [7] is not an option that will satisfy the job's SLO.

The other solutions shown in Table 1 propose running complex, and in some cases stateful, workloads on stateless cloud functions like AWS Lambda. Most of these frameworks (like Qubole's Spark on Lambda [36] and PyWren [24]) use an external storage platform such as Amazon S3 for the shuffle data. This allows individual Lambda functions to work as executors, relinquishing the CPUs as soon as the task is completed.

Since S3 is a multi-tenant service, there is a forced upper bound on the maximum number of I/O requests per S3 bucket. Although this may be relaxed as the number of buckets increases, the service usually tends to throttle when the aggregate throughput reaches a few thousands of requests per second. So, in general, associated workloads take a long time to finish, even though the overall I/O bandwidth is comparable to that of a local disk write. Also, even though the per-write cost is relatively low, workloads like CloudSort [9], which can trigger on the order of 10^{10} shuffle writes in single job execution, can incur enormous total S3 related costs. Note also that shuffle I/O may be reduced by employing data compression [13], but with added computational overhead.

Recently, [35] proposed to solve these problems by using a Redis cluster for storing intermediate shuffle data and S3 forthe initial input and final output. Redis, being an in-memory dictionary, significantly improves on I/O operations compared to disk writes, but is quite expensive as it requires the use of large VMs. Flint [26], another prototype of Spark on AWS Lambda, replaces AWS S3 with SQS [2] for intermediate data I/O using multiple distributed queues, which is a better fit for a high number of small writes. SQS does better in terms of throughput but is costlier and less reliable compared to AWS S3.

numpywren [38] is a system for linear algebra applications built on a serverless architecture. It is based on PyWren and relies on VMs for deploying an external scheduler and provisioner. The task queue is implemented on Amazon SQS and is constantly probed to change the number of executors to map to the current degree of parallelism of the job. SplitServe can inherently perform this task without needing any additional VMs. As with [24, 35, 36], numpywren also uses AWS S3 for external storage.

ExCamera [20] is a dedicated framework designed to perform digital video encoding by leveraging the parallelism of thousands of Lambda functions. It uses AWS S3 to perform intra-thread communication. It also relies on VMs for coordination and *rendezvous* between the Lambda functions.

Cirrus [25] is a machine-learning training framework which uses AWS Lambda for compute scalability and AWS S3 for high throughput storage. Also, a relatively small amount of high-performance storage to improve the performance of these pipelines is achieved by using a few VM instances to implement an in-memory parameter server. In some cases, Cirrus converges 3× faster than Tensorflow and 5× faster than Bosen [45]. While Cirrus can outperform frameworks which runs strictly on VMs in terms of training completion time, it does not outperform on cost, which may be up to 7× higher.

gg [19] is yet another framework that helps users execute burst-parallel applications – e.g., software compilation, unit tests, video encoding, or object recognition – by using thousands of parallel threads on a cloud function service,

like Lambda, to achieve near-interactive completion times. Even though in this work the framework relies on external ephemeral storage solutions like AWS S3, Redis and Google Cloud Storage, the authors claim to have found that two Lambda functions can communicate directly using off-the-shelf NAT-traversal techniques, at speeds up to 600 Mbps, with variable performance. Hence, if used in future, such technique can likely improve performance of systems using cloud function service for complex workloads.

FEAT [32] explores the idea of auto-scaling using Cloud Functions (CFs). It first launches an application on a pool of CFs then determines the number of VMs and the number of cores on each VM required by the application and subsequently launches these VMs in the background. When the VMs are ready, the control is transferred from the CFs. Note that FEAT considers workloads like publish/subscribe and request/response, which are largely stateless and scalable and thus a good fit for the serverless paradigm. In particular, these workloads do not face the problems of stragglers, intermediate data, or dependencies, and hence they do not need to work with external substrates for storage. Tasks of individual jobs are not, in any single point in time, divided between IaaS and FaaS services.

3 Background and Motivation

Background on Apache Spark: Apache Spark [39] is a widely used, open-source, general-purpose framework for large-scale, distributed data processing. A Spark cluster has a single master (driver) and one or more slaves (worker nodes, executors). A Spark driver is the entry point of a Spark application – it runs the main function of the application and is the place where the "Spark context" is created. A job is a unit of work that a user is interested in. Upon the submission of a job, Spark creates a series of stages. This breakdown is represented as a Directed Acyclic Graph (DAG) which is an "action plan" for the execution of the job. A stage may have multiple parallel tasks (*e.g.,* map, reduce). The Spark driver contains several components that are collectively responsible for converting a user submitted Spark application to an execution Directed Acyclic Graph (DAG) composed of multiple stages, each of which in turn consists of parallel running tasks. The driver schedules job execution and negotiates with the cluster manager if any (*e.g.,* Mesos, Yarn, Kubernetes or Spark's Standalone cluster manager) for resources in terms of "executors." An executor is a distributed agent responsible for the execution of tasks. Every Spark application has its own executor process(es). Spark offers two modes of executor allocation: static allocation, wherein all executors run for the entire lifetime of the application, and dynamic allocation, that lets an application start with a predefined minimum number of executors, which can grow in numbers (up to a specified maximum) as and when the resources becomes available in the cluster. If however, an executor is idle for some time, it is killed and the resources are returned to the cluster.

Spark creates stages at state transfer boundaries, *i.e.,* when the execution of the next stage depends on the output of the tasks from the previous stage. This movement of data between nodes at stage boundaries is known as *shuffling.* In static allocation of executors, the shuffle data can be kept in the executor's memory which can act as a server for other executors who want to access this data. However, in the case of dynamic allocation, since an executor might be killed due to inactivity, to prevent the intermediate shuffle data from being lost, all of the intermediate shuffle output is written to the local disk.

Why Combine VMs and Lambdas? From a tenant's perspective, two advantages of AWS Lambdas over IaaS VMs (simply VMs henceforth) are of significance to our work. First, Lambdas offer a finer-grain, pay-per-request billing model. Second, Lambdas offer significantly lower startup latency than VMs. Existing studies [10, 11, 32, 44, 49] and our own measurements show that an AWS VM may take up to 2 minutes or more after the user requests it to start working. While a "cold-start" Lambda (*i.e.,* requiring a fresh VM boot-up and container launching) incurs a similar startup latency, a "warm-start" Lambda (*i.e.,* an already existing recently-used VM with relevant code/data likely to be in memory) incurs a much smaller startup latency of only about 100ms [44].[4] Second, Lambdas are cheaper for short-lived resource usage patterns through sub-second minimum cost and billing granularity.[5] Specifically, AWS Lambdas are priced based on memory allocated (ranging from 128MB-3GB of memory per instance with one vCPU per 1.5GB) and execution time. Also, their cost is given by the product of memory allocated and time-in-use rounded up to the nearest 100ms (there are additional costs related to number of invocations). On the other hand, VM prices are typically based on their type (*e.g.,* reserved, on-demand, burstable, spot) as well as resource capacities (CPU, memory, storage). In particular, AWS VM cost is given by their price times the time-in-use rounded up to the nearest second.

Figure 1 compares the cost of one vCPU on a m4.large instance to an AWS Lambda with 1.5GB memory which gives it an effective capacity of one vCPU. AWS EC2 instances impose a minimum 1-minute charge on the users following which the cost is calculated in 1 second increments. In the figure, we can see this as a horizontal line till 60 seconds, and a step-wise monotonically increasing pattern following that. In comparison, AWS Lambda bills the tenants in 100 milliseconds increments with a minimum charge of only 100

[4] AWS keeps "dormant" Lambda alive for ~90 minutes.

[5] There are other cost advantages of Lambdas stemming from the fact that they relieve the user of managing OS and runtime environments. These cost aspects are harder to quantify in the context of our work and are beyond the scope of this paper.

milliseconds, presenting itself as a much more continuous-looking step function as soon as the billing cycle starts. The graph shows how quickly a Lambda can overshoot a VM in terms of cost of execution. Cost curves like this can help the tenants choose the duration for which they want their workloads to run on Lambdas while working under budget constraints.

Given these properties, it is natural for a cost-conscious tenant to use Lambdas as follows: instead of over-provisioning VMs to deal with workload mispredictions, the tenant can provision them closer to predicted requirements and use the agile Lambdas to reactively handle unexpected spikes in workload. Indeed this is the basic idea behind recent works such as [25, 32, 37, 49] which deal with stateless workloads.

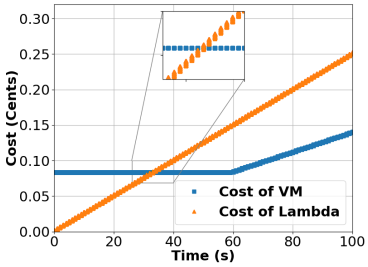

Figure 1. Cost of one vCPU on a m4.large instance compared with an AWS Lambda function with 1536 MB memory which gives an effective capacity of one vCPU.

Why SplitServe Is Challenging: In their current form, Lambdas have a number of restrictions that pose hurdles in adapting the above ideas for complex workloads:

- *Limited resource capacity:* Lambda containers do not possess enough memory for the needs of many workloads. We have found that due to their relatively modest memory allocation, garbage collection may begin posing significant overheads after only a few minutes of execution even for moderately memory-intensive workloads. Finally, each instance is provided a relatively small local storage (/tmp directory) of size 512MB to store intermediate state [4].
- *Limited lifetime and user control:* Lambdas are terminated after 15 minutes, rendering them unsuitable for longer-running jobs [4]. Also, a user cannot easily control the order in which interacting Lambdas are actually started by the provider. Control could be achieved either by an orchestrator on a VM, which would introduce delays, or by "step functions" that may increase costs [5, 17].
- *Poor support for sharing of intermediate state:* Since Lambdas do not expose an IP address L, there is no direct channel for an entity (whether VM or Lambda) to send data over the network to a Lambda after its initial invocation. This in combination with (i) their limited lifetime and (ii) the user's lack of control over when Lambdas are initiated by the provider, imply that state transfer between Lambdas must rely on a storage facility external to the source Lambda's container. Existing solutions (see Section 2) use either: (a) Amazon's S3 or SQS [26], which have been reported to be slow[6]; or (b) in-memory datastores such Redis [35], which are relatively expensive.
- *Steeper cost curve for longer-lasting work:* Currently, Lambdas charge a higher price per unit resource as compared to VMs which results in higher costs for long-lasting resource needs, as discussed above (see Figure 1).

A final challenge concerns the effort involved in modifying the existing framework codebase (written for VMs) to let tasks execute on Lambdas. Two observations are worth noting here. First, while this effort (and corresponding cost) may be non-trivial – *e.g.,* see details of our design in Section 4 – it would be amortized over the long run by the operational cost improvements offered by SplitServe. In fact we benefited from the Qubole project [36] that had ported Spark to run completely run Lambdas (except for the master). Second, this points to an important future research direction developing systems support for (largely) automating such IaaS to Lambda code modification.

4 SplitServe: Overview and Design

We intend for SplitServe to be part of a larger system that performs *cost-conscious autoscaling* of Apache Spark-based latency-critical workloads in the public cloud. As Figure 3 shows, we view such an autoscaling system as combining resource procurement/management at two time-scales: (a) *inter-job* decision-making spanning multiple jobs (how many VMs should the cluster have during the next few minutes?) and (b) *intra-job* decision-making (how to best satisfy the service-level objective (SLO) of an individual job based on resource requirement prescriptions made by (a)?) SplitServe's design and operation are concerned with the latter **intra-job** resource management.

4.1 Autoscaling Apache Spark: Inter-Job Management

A Scenario Motivating Splitserve: Suppose a budget conscious tenant runs a **long-term, latency-critical stream** of Apache Spark jobs. Also, the workload may be random and **not perfectly predictable**, either in terms of the job sizes or when jobs arrive. Multiple jobs can arrive together and may need to be executed simultaneously. To avoid wastage, a budget-conscious tenant would dynamically adjust the size of the VM cluster it procures from the cloud in response to such workload variations rather than always provisioning for the worst-case needs (note that, generally, the worst-case needs may themselves not be perfectly predictable). For such autoscaling, the tenant would need to estimate individual job execution times as a function of key workload properties such as relevant input data features (*e.g.,* size) as well as the inter-job arrival process. Much work has been done on such estimation and modeling using techniques spanning offline

[6] [27] attributes their slowness to their fault tolerance mechanisms that it deems excessive for the relatively short-lived intermediate state transferred.

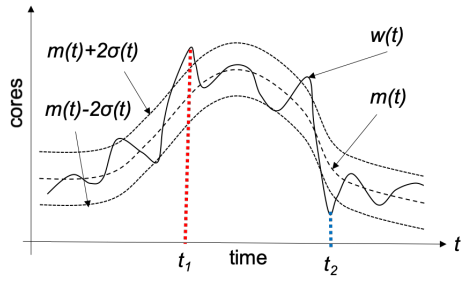

Figure 2. Illustrative example of average predicted workload needs (in terms of number of executors, one per core) with 95% confidence bands over a typical workday. Random $w(t)$ is the true number of executors/cores needed. Note that $w(t_1) > m(t_1) + 2\sigma(t_1)$, *i.e.,* additional executors are needed at time t_1 even if $m(t_1) + 2\sigma(t_1)$ VM-based executors are procured by the tenant at time t_1. Also, $w(t_2) < m(t_2) - 2\sigma(t_2)$, *i.e.,* VM-based executors will be idling at time t_2 even if only $m(t_2) - 2\sigma(t_2)$ VM cores are procured by the tenant at time t_2.

profiling and online learning, *e.g.,* [15, 30], that one may borrow from.

As an illustrative example of such autoscaling in Figure 2, suppose t is a particular time in a regular workday and let $m(t)$ and $\sigma^2(t)$ be the estimated mean and variance, respectively, of the number of Spark executors[7] needed to service the tenant's workload at time t. The tenant may choose to provision VM resources based on a high percentile of its estimated needs (*e.g.,* policy based on $m(t) + 2\sigma(t)$ in Figure 2). Despite such planning, it is possible that a job will arrive to find insufficient VM resources to meet its execution-time SLO. An example of this occurs at time t_1 in Figure 2. If new VMs are requested at t_1, they may not be available for a few minutes causing the SLO to be violated. If SLO violations are unacceptable, the tenant would need to provision more VMs (than based on $m(t) + 2\sigma(t)$) with associated higher costs. SplitServe would allow the tenant to overcome these delays and get additional resources at t more quickly via Lambdas. Equally importantly, SplitServe may allow the tenant to explore a less conservative inter-job VM procurement strategy (*e.g.,* policy based on $m(t)$ as opposed to $m(t) + 2\sigma(t)$ in Figure 2). Such a policy **may**[8] prove less costly but would result in more occurrences of VM shortfall to be bridged by Lambdas. Assuming the tenant employs a *cost manager* that determines a suitable combination of VMs and Lambdas per-job based on these considerations, we focus on designing SplitServe to run the job on its prescribed number of cores seamlessly.

[7]We assume, for simplicity, one core per executor with corresponding allocation of other resources within a VM. This can be generalized to more complex allocations per executor.

[8]Since Lambdas are more expensive than VMs on a per-unit resource price basis (besides having other shortcomings discussed earlier), lowering the number of procured VMs in this manner would generally prove useful only up to a certain point.

4.2 SplitServe Design for Intra-Job Management

SplitServe's design consists of three key facilities shown in Figure 3.

Launching Facility: SplitServe implements a "launching facility" that can share access to the system-wide VM/Lambda state with the cost manager. This state keeps track of where the executors for a job are currently running and which VM cores are currently free (if any). The launching facility arranges for the requested number of cores for a new job from the currently free cores and, if needed, by launching new Lambdas.

Segueing Facility: Besides the steeper cost of Lambdas beyond a certain duration (recall Figure 1), another reason for segueing longer-running tasks (initially started on Lambdas) to VMs is the relatively small memory allocation of Lambda containers. In particular, the smaller memory on Lambdas results in more frequent invocations of the JVM garbage collector (GC), which in turn hurts the overall workload performance. This may make it difficult, or in some cases impossible, to run a workload processing large datasets on a small number of Lambdas. SplitServe implements a "segueing facility" that launches VMs in the background matching the cores procured through any Lambdas that the launching facility starts. These VMs are only launched if the job's expected execution time (*i.e.,* the SLO conveyed by the inter-job manager) exceeds the nominal VM start up delay. Note that, for jobs with SLO smaller than the VM start up delay, starting new VMs would be futile.

State Transfer Facility: Many Spark workloads engage in significant data transfers across stages, *i.e.,* the shuffle operation. If an application framework dynamically allocates executors – which is the case for our cost-conscious tenant – then to prevent the loss of any shuffle data when dealing with fleeting executors, the shuffle data is written to local storage. This becomes a problem for Lambda-based executors, where (a) users have to pay a steeper price per unit resource than VMs to preserve the shuffle data, and (b) a relatively small amount (512 MB) of local storage is made available to each Lambda. Currently popular solutions to this problem are based on using a cloud-offered shared storage layer (*e.g.,* S3 or SQS in the case of AWS), possibly in conjunction with in-memory caches like Redis or Memcached. However, these solutions tend to be either too slow (for reasons such as throttling) or too expensive or both. Therefore, a final important facility SplitServe implements is a storage layer that enables fast transfer of state to and from Lambdas.

An Example: We use the example shown in Figure 3 to clarify how these facilities work together. At the very bottom of the figure, we depict the evolution of the VMs and Lambdas being employed by SplitServe in panels A-D. Each VM is

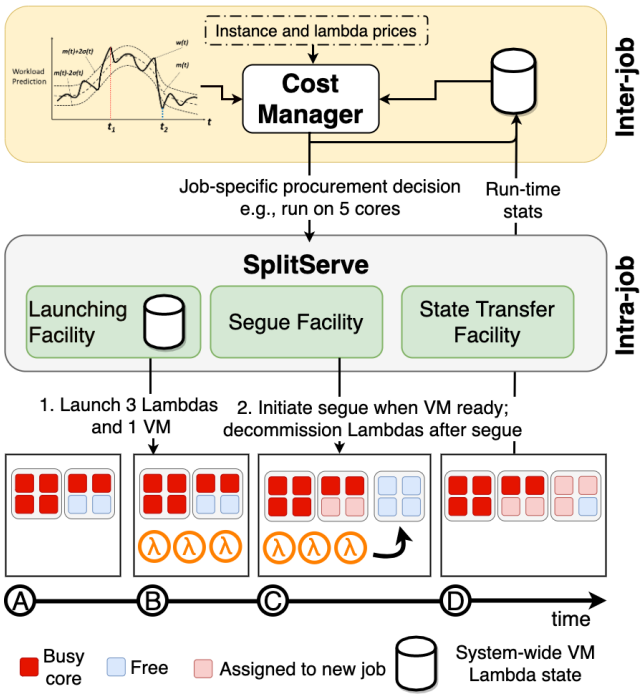

Figure 3. SplitServe operates at the individual job granularity and combines the agility of Lambdas with the longer-term lower costs of VMs. A complementary resource procurement system would carry out cost-aware resource procurement based on inter-job objectives and rely upon SplitServe to meet individual job's performance needs.

assumed to have 4 cores; the scheme to denote a core's occupancy appears in the legend. In our example, the inter-job cost manager (at the very top) reckons 5 cores to be appropriate for meeting a new job's SLO and conveys this to SplitServe along with the SLO itself. Upon receiving this request, SplitServe consults the system-wide VM/Lambda state to find only 2 VM-based cores currently available in the cluster (panel A). SplitServe then launches 5-2=3 Lambdas making a total of 5 cores available for the job (2 VMs and 3 Lambdas) and starts directing tasks to them (panel B). Let us assume that the job's expected duration is longer than the nominal VM startup delay. Given this, SplitServe also launches a new VM. Upon this VM becoming ready, SplitServe orchestrates seguing of the tasks running on Lambdas to 3 cores on this VM (panel C). Finally, once the tasks on the 3 Lambdas have completely transitioned to using cores on the new VM, the Lambdas are decommissioned bringing the system to an all VM execution (panel D). At each of these steps, SplitServe updates the VM/Lambda state. Alternatively, segue may occur to an executor that becomes available on an *existing* VM.

4.3 SplitServe Implementation Details

We implement SplitServe by modifying Apache Spark version 2.1.0 (rc5). We also adapt certain features from Qubole's Spark-on-Lambda [36], which is a rendition of Spark that uses AWS Lambdas for *all* the executors running a job.

Launching Facility: Our implementation of SplitServe's launching facility and the VM/Lambda state mainly spans Spark's *CoarseGrainedSchedulerBackend*, *StandAloneSchedulerBackend*, *CoarseGrainedExecutorBackend*, and *ExecutorAllocationManager* classes. The *ExecutorAllocationManager* class is responsible for requesting executors based on a job's requirements [50]. Since these requirements can change throughout the lifetime of the job, Spark allows the users to use the "dynamic allocation" mode where the number of executors allocated to a job can increase or decrease as the job makes progress. The *CoarseGrainedSchedulerBackend* class is responsible for identifying available executors, and *StandAloneSchedulerBackend* is responsible for launching new executors if needed. The *TaskScheduler* class assigns tasks to available executors. SplitServe needs the ability to distinguish an executor running on a Lambda vs. one running on a VM. Towards this, we add the functionality of launching both VM and Lambda executors to *StandaloneSchedulerBackend* and modify appropriate data structures to account for the two types. The *CoarseGrainedSchedulerBackend* class records the time whenever a Lambda-based executor is registered. This effectively allows SplitServe to keep track of the time since a job's beginning that a Lambda-based executor has been executing the job's tasks – this information is used for decision-making regarding launching new VMs and/or seguing the Lambda-based executor's tasks to VMs (as described next).

Segueing Facility: Everytime the *CoarseGrainedBackendScheduler* needs to pick an executor to schedule a new task upon, it first checks if there are any Lambda-based executors amongst the list of available executors, and if so, also checks how long they have been running for by comparing the current time against the timestamp recorded at executor registration. If a Lambda executor has been running for more than a pre-defined time threshold it is possible that it will either run into GC-induced slowdown or overrun its budget.

If we simply kill an executor at this stage, the tasks currently running on this executor will be marked as 'Failed'. Although Spark is designed to handle such failures with its recovery-from-fault mechanisms, the system enters into an "execution roll-back," which typically means a high recovery time (and corresponding costs) due to cascading recomputations. This is similar to working with transient resources as discussed in [46], and may greatly increase job execution times and create a large amount of network traffic. To overcome this, SplitServe simply stops directing additional tasks to a long-running Lambda-based executor. This makes the long-running Lambdas finish any pending tasks and get gracefully decommissioned once they become idle. We add a new configuration parameter *spark.lambda.executor.timeout* which acts as the threshold on which these decisions are based. Note that this is a configurable "knob" which can be set based

on concerns like budget, classification of big or small jobs, ability of the inter-job cost manager to spin-up new VMs, or an expectation that existing VM-based executors will soon become available.

HDFS-Based State Exchange for Lambdas: In Section 2, we discussed the storage solutions which have been explored by other state-of-the-art frameworks and offer a qualitative analysis of how and why these solutions are either not suitable or not feasible for our work.

SplitServe uses a single common high throughput storage layer, which can be accessed by both VM and Lambda based executors. **Our choice of HDFS is solely due to the relative ease of implementation compared to with alternatives.** Spark comes with library support for HDFS writes. We leverage this support to modify Spark internals to direct intermediate shuffle files to HDFS. However, SplitServe can use any other similar storage facility that offers the tenant a desirable cost-performance trade-off including ideas from recent research such as [27, 35], *etc.*

We modified various storage-related classes in SplitServe (like *BlockManager*, *DiskBlockManager*) to allow both VM-based and Lambda-based executors to write in one common place while following the Spark semantics of directory structure. Both VM- and Lambda-based executors use their uniquely identifiable and distinguishable Ids as an entry point into this directory structure. Using these unique Ids also allows the user to perform a fine-grained analysis of the work distribution between the two types of executors.

5 Experimental Evaluation

We first describe our experimental set-up including offline workload profiling[9]. We then provide a comparative performance evaluation of SplitServe vs alternatives. We use diverse latency-sensitive workloads from two well-regarded benchmark suites: Spark-SQL-Perf [40] and Intel HiBench [22] . From DataBrick's Spark-SQL-Perf, we present results for TPC's Decision Support (TPC-DS) ETL-type queries, which represent real world business critical analytics queries. From Intel HiBench, we present results for (i) WebSearch (or PageRank), which is compute and shuffle I/O intensive, and (ii) Distributed K-Means, a compute intensive machine learning-based workload with some shuffle I/O. Finally, we also give results for concurrent calculation of Pi – although not latency-sensitive per se, we use it as a proxy for a workload that is different from the previous ones in being compute-intensive, but having very little shuffling.

[9]In practice, offline workload profiling (to trade off cost and performance when autoscaling) can be continually updated by online reinforcement learning, *e.g.,* [30], *i.e.,* when jobs are repetitive as in Figure 2. Such profiling results, service pricing, and runtime execution progress information may be used by a cost manager to control the invocation of SplitServe.

5.1 Workload Profiling

Recall mention of our interest in latency-critical workloads in Section 1. Broadly speaking, any workload where a response time greater than the start-up time of VMs is an acceptable SLO is not considered as latency critical. In such cases, autoscaling solutions using only VMs can be adopted, and hence such workloads are outside the scope of our work. Workload SLOs and how SplitServe considers them are discussed in Sections 3, 4.

For a latency-sensitive Spark job to meet its desired execution time target, its degree of parallelism and resource allocation for individual tasks must be carefully chosen. We carry out offline profiling for our workloads to understand how their job execution times depend on (a) input data size and (b) degree of parallelism (*i.e.,* number of executors).[10] As the number of executors is increased, the input data size (or task size) per executor decreases proportionately. We illustrate our profiling methodology and findings for PageRank; similar ideas and findings apply to our other workloads as well. Throughout this study, we assign each executor exactly one core. Consequently, the term "executor" is synonymous with "one core" in what follows. Similarly, we only consider homogeneous task partitioning (*i.e.,* the tasks of a job are of the same size). Alternate resource allocation policies (*e.g.,* multiple or different number of cores per executor) or heterogeneous task sizing constitute complementary directions that we do not explore. Recall that an AWS Lambda is assigned at most 1.5 GB memory per core, which implies that there is typically more memory available for a VM-based executor than a Lambda-based executor. In our illustration here, we consider three input dataset sizes for PageRank jobs: "large" (100,000 pages), "medium" (50,000 pages), and "small" (25,000) pages. We also consider much larger input datasets in the next subsection.

Profiling Lambda-based Executions: We report a first set of experiments on SplitServe wherein the executors are **all** Lambda-based. We run the master on an adequately provisioned m4.xlarge AWS VM. As seen in Figure 4(a), our profiling offers a classic "U-shaped" curve expected for the execution time of a parallel workload [29, 33]. The curve suggests that, for a fixed input dataset size, there exists a "performance-optimal" *degree of parallelism*: increasing the degree of parallelism further results in poorer performance owing to the communication overheads while lower degrees do not fully extract the benefits of parallelism. We also plot the total cost (on the Y2 axis) which depends both on the number of executors and on the total execution time. With these profiles, decisions of the following type can be made: in case of a "large" PageRank job, if the execution time needs to be less than 70s, then two executors would be the lowest-cost

[10]Given a continual job stream as in Figure 2, such profiling can similarly be done online.

choice; however, if the execution time needs to be less than 60s, then the only choice is 4 executors.

Profiling VM-based Executions: We perform a second set of experiments using all VM-based executors for which we employ "vanilla" Spark (worker nodes are EC2 instances with types determined as explained momentarily) with the master again running on a m4.xlarge instance. For each degree of parallelism, we use the fewest number of instances that provide the required number of cores to minimize the inter-VM communication overhead: m4.large, m4.xlarge, m4.2xlarge, m4.4xlarge, m4.8xlarge, m4.16xlarge, and two m4.16xlarge, for 1-2, 4, 8, 16, 32, 64, and 128 cores, respectively. Note that, generally, the cost per core grows with VM size whereas execution time reduces because of the substantially less inter-VM communication, *e.g.,* for data shuffling – see the cost curves of Figure 4(b). Also, note from Figure 4(b) that, even though the optimal degree of parallelism is the same as that when running only with Lambda-based executors, the overall execution time for the job is much lower when running on VMs (as expected).

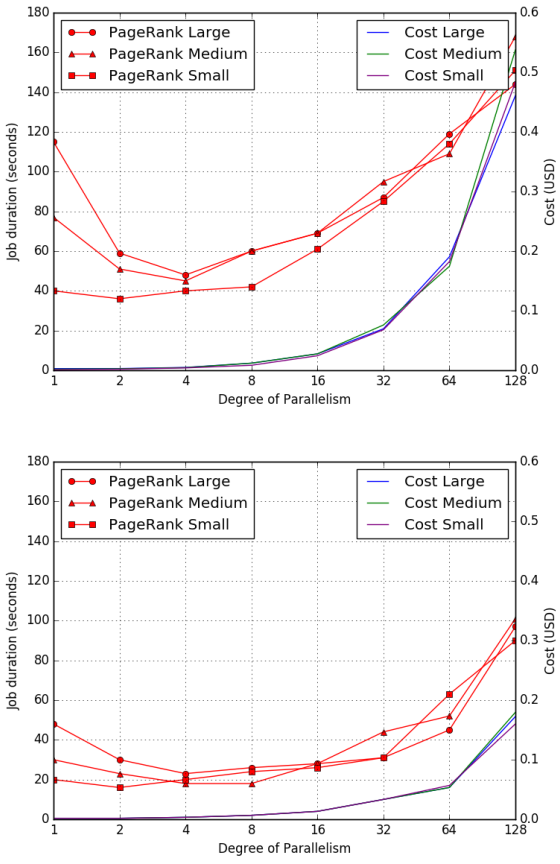

Figure 4. Offline profiling to determine cost and performance vs. degree of parallelism for PageRank jobs when using (a) only Lambda-based executors or (b) only VM-based executors.

Metrics and Scenarios: We compare several scenarios for processing an incoming Spark job in terms of *performance* (execution time, which is also the time between job submission and completion since SplitServe does not queue jobs) and *cost* (incurred towards any VMs, Lambdas, and storage systems such as S3 involved in executing the job). **Note that we only report the cost incurred towards the job in question under the scenario being considered. Specifically, this cost should not be confused with or viewed as being a proxy for the long-term cost incurred by the tenant (*e.g.,* its monthly cloud bill). The latter would depend on the autoscaling policy especially the degree of VM under/over-provisioning – recall Figure 2.** Each scenario represents a combination of (i) whether adequate resources (we simply use "cores" henceforth although it should be clear we are also referring to other resources such as memory associated with them) are available, (ii) when enough cores are not available, whether autoscaling is done, (iii) whether the ability to run on Lambdas exists (as with Qubole-on-Lambda and SplitServe but not with vanilla Spark), (iv) when Lambda-based execution is possible how shuffling is done (S3 vs. HDFS), and (v) whether seguing from Lambdas to VMs is possible (as in SplitServe but not in Qubole-on-Lambda). Below we describe each of our scenarios. The scenarios involving (vanilla) Spark or Qubole represent "baselines" representing the state of the art. We use R to denote the arriving job's required number of cores. In scenarios where the number of cores currently available in the cluster is less than R, we denote it by r and the difference $R - r$ by Δ.

Spark r VM: The job arrives to find the cluster under provisioned and the system does not employ autoscaling. Specifically, the job is executed by vanilla Spark with $r < R$ cores available on VMs. The job runs using only these r cores for its entire execution.

Spark R VM: The job arrives to find adequate cores already provisioned on VMs. Vanilla Spark with R cores available on VMs. This obviously represents the best case from both a cost and a performance point of view without autoscaling (and so is extraneous to the problem SplitServe is trying to address).

Spark r/R autoscale: The job arrives to find the cluster under-provisioned (vanilla Spark with initially $r < R$ resources available on which the job starts running). Subsequently, upon observing/predicting that the job execution time is more than a threshold, after time t, $R - r$ additional cores are procured.

Qubole R La: The job is executed entirely using Lambdas. Specifically, the job runs on Qubole's Spark-on-Lambda, which launches executors on R Lambdas (each with one core), and uses Amazon S3 as external storage to store all the intermediate shuffle data.

SS R VM: The job arrives to an adequately provisioned system using SplitServe. That is, R cores are available on VMs

when the job arrives. We would like this to perform comparably to "Spark R VM." Any difference between the two would be indicative of overheads posed by SplitServe.

SS R La: The job arrives to SplitServe with no cores available on the VMs. All R executors are launched on Lambdas.

SS r VM / Δ La: The job arrives to SplitServe with $r < R$ cores available on VMs. The remaining $R - r$ cores are launched immediately on Lambdas but no seguing is performed from Lambdas to newly-available/procured VMs.

SS r VM / Δ La Segue: The job arrives to SplitServe with $r < R$ cores available on VMs. The remaining $R - r$ executors are launched immediately on Lambdas. Additionally, after a time threshold t, which is smaller than the predicted execution time of the job, Δ cores are made available on VMs (either procured or became free); so, the execution flow is segued from Lambdas to the newly-available VM resources.

5.2 Evaluation Results for Our Workloads

We begin by describing our findings for TPC-DS, a decision support query benchmark used to business critical analytical queries, from a purely performance perspective. We show how SplitServe, our VM+Lambda hybrid solution, outperforms existing pure Lambda based solutions as well as improving upon VM based autoscaling solutions. We then explore our findings from Intel HiBench, running WebSearch and Machine Learning benchmarks and show that in resource constrained scenarios where a hybrid solution may not be the most optimal (in terms of cost, performance or both) due to fewer VMs available than desired, an all-Lambda solution may be a better choice to consider, and that all-Lambda solution under SS outperforms other baselines.

TPC-DS: TPC-DS is a decision support question benchmark that models several generally applicable aspects of a decision support system. The queries generally have diverse compute and I/O footprints across them and are a prime example of ETL workloads. Specifically, we used DataBrick's Spark-SQL-Perf, a benchmark to test Spark's SQL performance. The TPC-DS workload suite consists of 100 queries, out of which we picked 10 with a range of compute and memory requirements and are I/O intensive (*i.e.,* heavy on shuffle data generation) and tested them over a range of scaling factors. These queries allows the benchmark to capture important aspects of the interactive, iterative nature of on-line analytical processing (OLAP) queries many of which are constructed to answer immediate and specific business questions, hence denoting their latency critical nature. Out of those, we present the results of 4 queries (Q5, Q16, Q94 and Q95) which were run on a scale factor of 8. The workload is run on 32 cores using a m4.10xlarge instance to launch VM based executors. Since we want to match the performance of different systems as closely as possible, we run the SplitServe Master and HDFS on a m4.10xlarge instance as well to get similar dedicated EBS bandwidth.

We use $R = 32$ and $r = 8$. Starting with "Spark 8 VM" results in Figure 5, we observe that running the queries on only a subset of desired resource requirements can deteriorate performance by up to 4×, or in some cases even more than that. Running on "Quobole 32 La" takes 21.7× more execution time on average[11]. "SS 32 VM" compares closely with "Spark 32 VM" performing at par in most cases and doing only 1.6× poorer in the worst case. Since there is a large amount of intermediate shuffle data to be transferred over network, Lambda's unreliable and proportional to memory network bandwidth proves to be a bottleneck for SS 32 Lambda in the worst case performing ~2.3× poorer than "Spark 32 VM". Since "SS 32 VM" performs very similarly to "Spark 32 VM", combining VMs and Lambdas proves advantageous. As more tasks are pulled to faster VM based executors, we see a continuously improving performance. **This shows the efficacy of having a hybrid solution for executing analytical and latency critical workloads.** Unlike many other frameworks, SplitServe is able to scale as required using Lambdas, able to run the queries (due to being independent of slow external storage like S3) and does not rely on specialized storage solutions (like Redis) to finish execution in required time. On average, "SS 8 VM / 24 La" addresses insufficient resources much more efficiently and takes 55.2% less execution time compared to VM based autoscaling. Since most of these queries finish executing under, or in some cases at about, 60 seconds, no tasks needed segueing from Lambdas to VMs.

WebSearch: PageRank is an algorithm used by Google Search to rank web pages in their search engine results. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important a website is. The underlying assumption is that, more important websites are likely to receive more links from other websites. Specifically, we used Intel HiBench's WebSearch (PageRank) workload. This workload spends most of its time on iterations of several CPU-intensive tasks with moderate disk I/O and memory utilization (but considerably more than distributed K-means clustering). We run this workload with a data set size of 850,000 pages. The experiment involved 16 cores/executors of m4.4xlarge EC2 under Vanilla Spark.

Running on only $r = 3$ cores, instead of $R = 16$, results in a performance degradation of around 2.1×. Even with VM based scaling, total execution time is worse by as much as 2×. Since PageRank is much more shuffle intensive than K-means clustering, we see the effects of large amounts of shuffling becomes more apparent. Since Qubole's Spark-on-Lambda uses S3, the overall execution time increases by more than 60%, but SplitServe's HDFS based shuffling increases it by only 27%.

[11] Note that we were not able to get results on Q5 for Qubole's Spark-on-Lambda since their prototype encounters fatal errors while running this query.

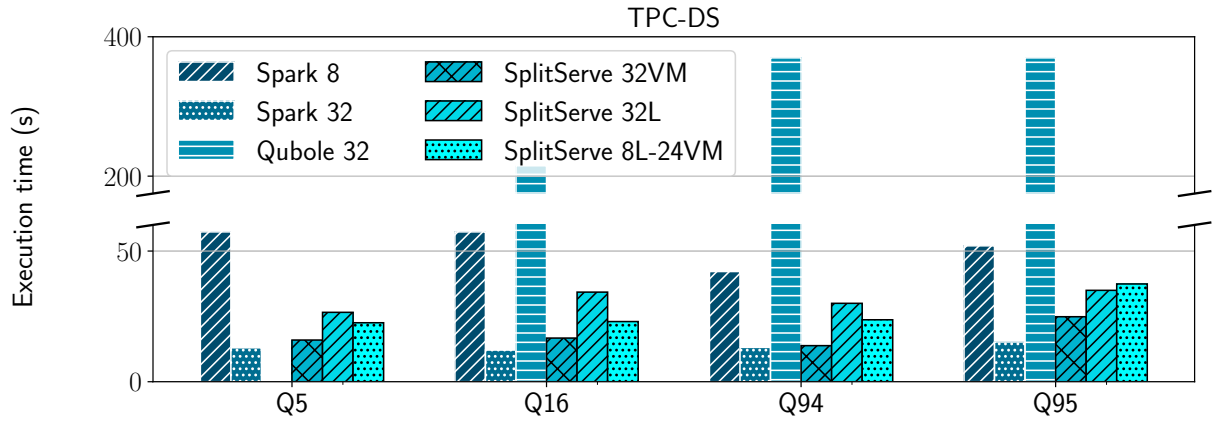

Figure 5. Comparing performance of Q5, Q16, Q94 and Q95 queries from Spark-SQL-Perf's TPC-DS workload suite.

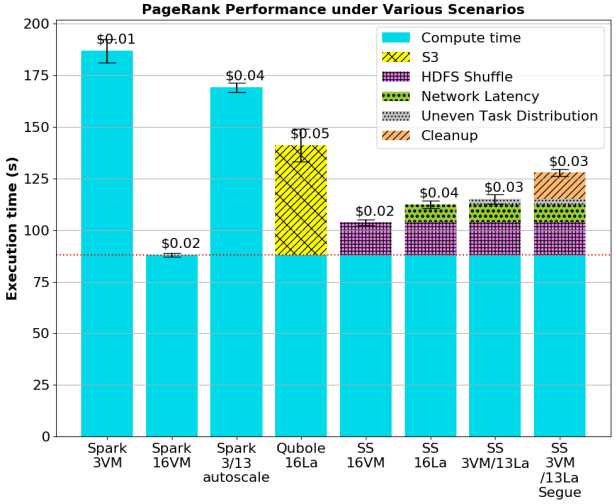

Figure 6. Comparing PageRank performance on SplitServe and other systems under various scenarios.

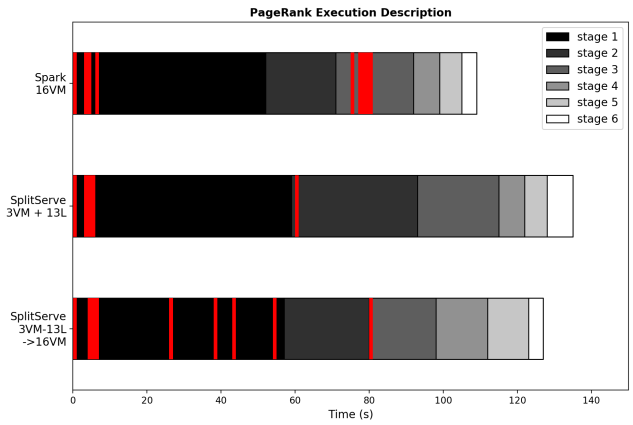

Figure 7. Comparing PageRank (6 execution stages) timelines for: (i) Vanilla Spark on a 16-core/executor VM; (ii) SplitServe with 3 VM cores and 13 Lambdas, and (iii) SplitServe of (ii) with segue to 16 VM cores. A thinnest red bar indicates when one new executor starts to be used. The blue bar indicates when segue commences – in this example, we suppose that a core on an *existing* VM became available at 45s, *i.e.,* earlier than the typical availability time if the core was on a VM newly requested at time 0.

In our setup for this experiment, the (single) HDFS node shares resources with the Spark master – we colocate both of these on a m4.xlarge instance with only 750 Mbps dedicated EBS bandwidth. On the other hand, in "Spark 16 VM" the executors are running on a m4.4xlarge machine with 2,000 Mbps of dedicated EBS bandwidth. Hence, the worker nodes provisioned on m4.4xlarge get about 3× more dedicated EBS bandwidth as compared to the Master node provisioned on a m4.xlarge machine. In our experiments, we trade off performance with cost-savings to show the efficacy of our system even with stringent budgets. Similarly, other overheads such as network latency in case of Lambdas and "clean-up" overhead in case of segueing to VM based executors are amplified due to the increased shuffle data traffic. Even in a conservative setup, executing the workload over both VMs and Lambdas under SplitServe offers about 32% improvement on overall execution time when compared to VM based scaling. Combining the joint execution with segue, we still see a performance improvement of 24% along with a cost benefit of 8%. A smaller cost improvement is an outcome of using a smaller instance (in terms of resources) on which to locate master node. Since the master is a longer running entity, a cluster manager should assign it to a core of one of its largest VMs when the application is I/O intensive.

In Fig. 7, we compare execution timelines for 3 scenarios.

Machine Learning Workload: K-means clustering is a widely used unsupervised learning technique that groups unlabeled data points into $k > 1$ clusters. An example of latency-critical K-means is for supervised attack detection, *e.g.,* [16], *i.e.,* requiring immediate and continual application on the latest online data for timely attack detection, and immediate updating (online K-means) once zero-day (previously unknown) attacks are identified. The algorithm works iteratively to (a) assign each data point to one of the k groups based on the features that are provided (map), and then (b) compute a new cluster center for each group (reduce). Data points are clustered based on their feature similarity.

We run the Intel HiBench ML K-means workload on a data set of size of 3×10^6 points, where each point is a 20-dimensional feature vector, and with $k = 10$ groups. Each job runs for a maximum of 5 iterations and tries to achieve a convergence distance of 0.5. Using workload profiling described in Section 5.1 we choose a degree of parallelism of 16 for our job which would allow it to meet a desired execution time of < 2 minutes for "Spark 16 VM." We use $R = 16$ and $r = 4$.

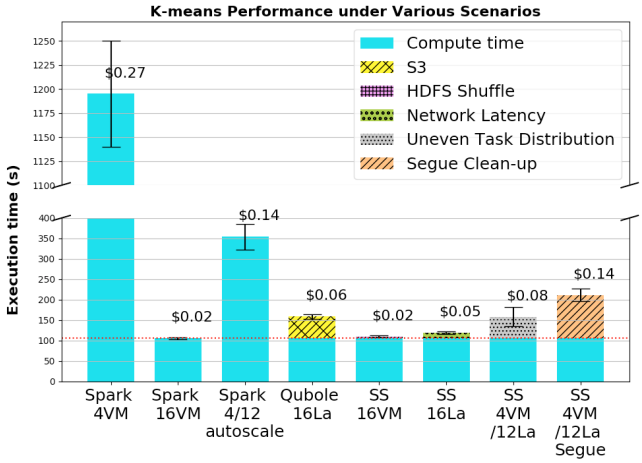

Figure 8. Performance and cost of our K-means clustering job under different scenarios. The confidence error bars are one sample standard-deviation from 15 independent trials. Horizontal red dotted line is the "Spark 16 VM" baseline.

In Figure 8, we report the comparative performance and cost of our K-means job. We see that running the same K-means job on only a subset of desired resources, *i.e.,* 4 executors (instead of 16), degrades the overall job execution time by a factor of 10×. Further, even with cluster size scaling, we see that the job still takes as much as 3.3× more time when compared to "Spark 16 VM" Even though the VMs are available to use within ∼1 minute, the slowdown is due to the fact that a large fraction of the tasks have already been scheduled on the existing executors which are overloaded and cannot be dynamically migrated to the newly available executors. Due to these queuing issues, VM-based scaling may not be a good option for a latency-critical job. Since K-means is compute as

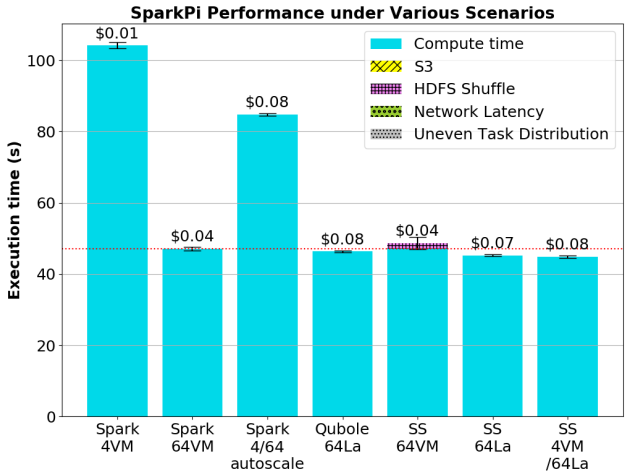

Figure 9. Comparing SparkPi performance on SplitServe and other systems under various scenarios.

well as somewhat I/O intensive, we see the effects of shuffling when running the job over Qubole's Spark-on-Lambda which shuffles over S3: it takes about 51% more time to finish the job than "Spark 16 VM" When comparing with SplitServe, we see that with an all-VM setup, we perform almost as well as "Spark 16 VM" even with an external shuffle over HDFS running on a node with a relatively modest I/O bandwidth. When we run the same job on SplitServe with only Lambdas, we do only 11% worse than "Spark 16 VM" As discussed in the previous subsection, the extra time is attributed to the fact that HDFS is running on a m4.xlarge machine (recall performance vs. cost savings trade-off). The slowdown is not significant since because *distributed* K-means clustering is not very shuffle intensive.

This experiment is another example where a hybrid solution performs poorer and costs more than an all-Lambda solution and hence opting for an all-Lambda solution under SplitServe gives much better performance.

SparkPi: An example of a compute-intensive application, it approximates the value of Pi by performing a Monte-Carlo simulation. This is done by throwing n random darts on (selecting n random points in) a plane, upon which there is a circle of unit area. The value of Pi is then approximated by calculating the fraction of points which fell in the unit disk. This job is highly parallelizable by giving almost an equal number of tasks (darts to throw) to each executor in the cluster and finally accumulating the result by performing a simple count. Since *count* is basically a *reduce* operation, there is negligible shuffling involved. Hence, SparkPi is an example of purely compute-intensive workloads, with negligible memory footprint or I/O overhead. In our experiments, we approximate the value of Pi by generating 10^{10} random points and running the job on 64 executors. We use a m4.16xlarge VM as the worker node to run these executors.

In Figure 9, we can see how the various baselines work under different scenarios. We start by running the job on a Vanilla Spark cluster with the best possible case, i.e., the job finds the required resources available in the cluster. Comparing this with the case where only a portion of the required resources (4 executors) are available in the cluster, it can be seen that the job has taken more than twice as long to complete. For SplitServe with an all-VM executors setup (*i.e.,* all the executors run only on VMs and not on Lambdas), it can be observed that the performance is similar to that of Vanilla Spark. Using Lambda executors, we see that both Qubole's Spark-on-Lambda and SplitServe's all-Lambda setup give similar performance to that of Vanilla Spark. This is mainly due to the fact that there is no shuffling involved in this workload. Finally, a more interesting case is when we split the work across both VMs and Lambdas. Even here we see a similar performance to that of our (best) baseline. Again, we did not assess the Lambdas-segue-to-VMs setup under SplitServe because the job finished under 1 minute.

6 Splitserve Discussion

SplitServe dynamic parameter selection: As discussed in Sections 4 and 5, selecting parameters ("knobs") for SplitServe largely depends on the operational conditions including overall budget (which dictates factors like how many VMs/CFs can be procured, how long can these resources be used for, etc.), resource availability expectation, desired performance/SLO, and the type of workload being run.

Figures 1, 2 and 4 show how some of these factors evolve over time and resources. The values chosen for these parameters in our work were largely shaped by the aforementioned factors, particularly the type of workloads we present in this paper. To offer a comprehensive evaluation for the reader, we chose workloads which encompass the large class of common ETL workloads (e.g., decision queries, machine learning, I/O intensive jobs, CPU intensive jobs) to give a better idea of how these parameters will affect a given class of workloads.

Recall footnote 9 where we mention how users can leverage offline profiling as a possible tool to tune the knobs provided in SplitServe to best suite their requirements. In future work, we will evaluate other methodologies where users can automate the process of tuning these parameters both statically and dynamically.

Comparisons of the costs of autoscaling: We point out that, generally, VM-based executors are less expensive than comparably provisioned Lambdas per unit time × resource. This is natural considering Lambdas are priced at a finer granularity and can be spun-up (especially warm start) and released more quickly. Regarding Figure 2, we discuss allocating VMs conservatively (paying more "global" cost), or otherwise, based on time-of-day predictions. Obviously, additional VMs in a conservative approach would cost more but there would be less autoscaling. As mentioned above, we only report the *marginal* cost incurred for the job in question under the autoscaling scenario being considered since "global" cost comparisons of this sort would vary greatly depending on specific streaming workloads, its volume, differences across policies, and the prices of VMs and Lambdas procured/released over time, all of which detract from the point of this paper: *how best to autoscale latency-critical workloads, considering associated (marginal) costs, with cloud functions.*

How to use SplitServe? To reiterate, SplitServe has been designed to efficiently run for a large class of widely used workloads (particularly of the ETL type). it is hard to comprehensively illustrate how SplitServe would perform across a long running streaming workload composed of various kinds of jobs under different long-term resource procurements. On the other hand, showing SplitServe's performance across only one class of workload (as some other works have) wouldn't rightly convey its capabilities and shortcomings.

To simplify exposition, in Section 5 we discuss Metrics & Scenarios and Workload Evaluation. The former topic represents the various scenarios a job could find itself in on arrival while the latter matches the scenarios to a class of workload and describes how SplitServe would perform autoscaling (when needed) for a given combination of {scenario, workload} under consideration.

7 Conclusions and Future Directions

We presented the design and implementation of SplitServe, an enhancement of Apache Spark, that can concurrently run a subset of the tasks within a parallel job on AWS Lambdas with the rest on VMs (the latter being the default). Thus, SplitServe is a valuable tool for a tenant interested in saving cloud costs by avoiding over-provisioning of VMs in the face of dynamic workloads. When newly requested VMs, or executors on existing VMs, do become available, SplitServe is able to move ongoing work from Lambdas to them. Our experimental evaluation of SplitServe using four different workloads shows that SplitServe improves execution time by up to (a) 55% for workloads with small to modest amount of shuffling, and (b) 31% in workloads with large amounts of shuffling, when compared to only VM-based autoscaling.

Generally, an executor assigned a certain number of cores on a VM vs. a Lambda-based executor with the same number of cores will have access to different capacities of other resources (*e.g.,* memory, IO bandwidth). In future work, we will explore the use of different task sizes for VMs and and Lambdas for better task-level load balancing. We will also devise SplitServe versions of other popular application frameworks, *e.g.,* Flink [8].

Acknowledgments

This research was supported by a NSF CSR grant 1717571, Cisco URP gift, and AWS credits gift.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Amazon [n.d.]. Amazon SQS. https://aws.amazon.com/sqs/.

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

[4] Amazon [n.d.]. AWS Lambda Limits. https://amzn.to/2vH102F.

[5] Amazon [n.d.]. AWS Step Functions. https://aws.amazon.com/step-functions/.

[6] Azure [n.d.]. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[7] A. F. Baarzi, T. Zhu, and B. Urgaonkar. 2019. BurScale: Using Burstable Instances for Cost-Effective Autoscaling in the Public Cloud. In *Proc. ACM SOCC*.

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[9] Cloud Sort [n.d.]. Sort Benchmark. http://sortbenchmark.org/.

[10] Yan Cui. Aug. 28, 2018. Cold start / Warm start with AWS Lambda. https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/.

[11] Yan Cui. Jan. 17, 2018. I'm afraid you're thinking about AWS Lambda cold starts all wrong. https://theburningmonk.com/2018/01/im-afraid-youre-thinking-about-aws-lambda-cold-starts-all-wrong/.

[12] Databricks Spark [n.d.]. Databricks Spark Optimized Autoscaling. https://databricks.com/blog/2018/05/02/introducing-databricks-optimized-auto-scaling.html.

[13] A. Davidson and A. Or. 2013. Optimizing shuffle performance in Spark. https://pdfs.semanticscholar.org/d746/505bad055c357fa50d394d15eb380a3f1ad3.pdf.

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[15] C. Delimitrou and C. Kozyrakis. 2016. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *Proc. ASPLOS*. Atlanta.

[16] H.M. Demoulin, I. Pedisich, N. Vasilakis, V. Liu, B.T. Loo, and L.T.X. Phan. July 2019. Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FINELAME. In *Proc. USENIX ATC*.

[17] Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. 2018. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. *CoRR* abs/1811.09721 (2018).

[18] firecracker-web [n.d.]. Introducing Firecracker, a New Virtualization Technology and Open Source Project for Running Multi-Tenant Container Workloads. https://aws.amazon.com/about-aws/whats-new/2018/11/firecracker-lightweight-virtualization-for-serverless-computing/.

[19] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. https://www.usenix.org/conference/atc19/presentation/fouladi

[20] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

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

[22] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. *Proceedings - International Conference on Data Engineering*, 41 – 51. https://doi.org/10.1109/ICDEW.2010.5452747

[23] IBM [n.d.]. IBM Cloud Functions. https://cloud.ibm.com/openwhisk/.

[24] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proc. ACM SOCC*.

[25] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019).

[26] Y. Kim and J. Lin. 2018. Serverless Data Analytics with Flint. https://arxiv.org/pdf/1803.06354.pdf.

[27] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 427–444. http://dl.acm.org/citation.cfm?id=3291168.3291200

[28] Danny Krizanc and Anton Saarimaki. 1996. Bulk Synchronous Parallel: Practical Experience with a Model for Parallel Computing. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*. IEEE Computer Society, Washington, DC, USA, 208–. http://dl.acm.org/citation.cfm?id=882471.883319

[29] R. Li, P. Guo, B. Hu, and W. Hu. Nov. 2019. Libra and the Art of Task Sizing in Big-Data Analytic Systems. In *Proc. ACM SoCC*. Santa Cruz, CA, USA.

[30] H. Mao, M. Schwarzkopf, S.B. Venkatakrishnan, Z. Meng, and M. Alizadeh. [n.d.]. Learning Scheduling Algorithms for Data Processing Clusters. https://arxiv.org/pdf/1810.01963.pdf.

[31] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, 1049–1058.

[32] J.H. Novak, S.K. Kasera, and R. Stutsman. 2019. Cloud Functions for Fast and Robust Resource Auto-Scaling. In *Proc. IEEE COMSNETS*.

[33] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. 2013. The Case for Tiny Tasks in Compute Clusters. In *Proc. USENIX HotOS*.

[34] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 1138–1149.

[35] 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 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[36] Qubole [n.d.]. Qubole Announces Apache Spark on AWS Lambda. https://www.qubole.com/blog/spark-on-aws-lambda/.

[37] J. Raj, M. Kandemir, B. Urgaonkar, and G. Kesidis. July 2019. Exploiting Serverless Functions for SLO and Cost Aware Tenant Orchestration in Public Cloud. In *Proc. IEEE Cloud*. Milan.

[38] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *CoRR* abs/1810.09679 (2018). arXiv:1810.09679 http://arxiv.org/abs/1810.09679

[39] Spark [n.d.]. Spark. spark.apache.org.

[40] Spark-SQL-perf [n.d.]. Spark-SQL-Perf Benchmark. https://github.com/databricks/spark-sql-perf.

[41] Splitserve [n.d.]. Splitserve. https://github.com/PSU-Cloud/splitserve.

[42] Xinhui Tian, Rui Han, Lei Wang, Gang Lu, and Jianfeng Zhan. 2015. Latency critical big data computing in finance. *The Journal of Finance and Data Science* 1, 1 (2015), 33 – 41. https://doi.org/10.1016/j.jfds.2015.07.002

[43] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis. June 2017. Using Burstable Instances in the Public Cloud: When and How?. In *Proc. ACM SIGMETRICS, Urbana-Champaign, IL*.

[44] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proc. USENIX ATC*. Boston.

[45] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2015. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 381–394. https://doi.org/10.1145/2806777.2806778

[46] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. 2016. TR-Spark: Transient Computing for Big Data Analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 484–496. https://doi.org/10.1145/2987550.2987576

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[48] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proc. USENIX HotCloud*.

[49] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proc. USENIX ATC*. Renton, WA.

[50] Zhi Zhou, Fangming Liu, Hai Jin, Bo Li, Baochun Li, and Hongbo Jiang. 2013. On Arbitrating the Power-Performance Tradeoff in SaaS Clouds. In *Proc. of IEEE INFOCOM*.