Off-the-shelf Data Analytics on Serverless

Michael Wawrzoniak[1], Gianluca Moro[1], Rodrigo Bruno[2], Ana Klimovic[1], Gustavo Alonso[1]
[1]Systems Group, Dept. of Computer Science, ETH Zurich
[2]INESC-ID/Técnico, U. Lisboa

ABSTRACT

Serverless has captured the interest of researchers and practitioners alike, being often considered the next step in the evolution of the cloud. Existing research, however, indicates it is ill-suited to data analytics due to the limitations of commercial platforms. This has led researchers to either design data analytics systems that work around the limitations of serverless platforms, suggest alternative serverless platforms, or both. In this paper we demonstrate that there is a third option: to provide the functionality needed to run off-the-shelf distributed data processing systems on top of existing serverless platforms (e.g., AWS Lambda) in a transparent manner. In the paper we discuss how this can be done and present initial experimental results of the TPC-H benchmark of unmodified Apache Spark and Apache Drill running on AWS Lambda. The results enable research in serverless data analytics that go beyond patching the shortcomings of existing commercial solutions and can be the basis for turning serverless into a general purpose computing platform.

1 INTRODUCTION

Serverless (most commonly offered in the form of Function as a Service, FaaS) has attracted significant attention due to its advantages [36]: fast start times, dynamic and automatic elasticity, fine-grained billing, and hands-off infrastructure management. The interest is such, that it has been claimed it could be the next step in the evolution of cloud computing [37].

However, there is a growing amount of work indicating serverless is ill-suited for data analytics due to limitations of the current platforms [23, 42]. Limitations often discussed in the literature include short function lifetime, lack of persistent state [19, 22, 39] and direct communication, as well as higher cost per second when compared to running on VMs [30, 31].

Motivated by these limitations, there is a growing amount of research on serverless data analytics along two main lines. One approach involves extending existing serverless platforms, often with VM-based services, to better support data analytics workloads on serverless infrastructure [27, 34, 39, 45]. Another approach is to build new engines specifically designed to work around the limitations of existing serverless platforms[26, 30, 31, 47]. All these efforts provide valuable insights on serverless data analytics but also implicitly give up on the many existing distributed data processing platforms (e.g., Spark, Flink, Drill, etc.) and depart from actual commercial offerings for serverless. In practice, this amounts to having to completely redesign data processing engines from scratch and/or building on the assumption that open source serverless platforms will be more efficient and feature-complete compared to the native serverless offerings from cloud providers.

In this paper we explore and propose an alternative solution for data analytics on serverless: to provide the functionality required to run off-the-shelf distributed data processing platforms (e.g., Spark or Drill) on top of existing commercial serverless platforms (AWS Lambda [1]). Although practically challenging, doing so is based on two insights: that serverless function platforms can be extended without modifying the underlying infrastructure, and that contemporary publicly available serverless functions are actually similar in memory size and computational power to what VMs were at the time when most popular data analytic platforms were initially developed (see Section 5). To test the feasibility of the idea, we introduce a minimal interposition layer between engine processes and the function platform to transparently provide the environment required by the engines. We use a further development of Boxer [44], able to handle unmodified data processing engine processes, to make the function environment appear like standard networked containers or VMs to the engine processes. The mechanism does not involve any changes to the serverless platform and provides a complete interface that enables running unmodified distributed data engines.

With our prototype, we are able to execute the TPC-H benchmark on unmodified Apache Spark [48] and Apache Drill [5] running on AWS Lambda. We demonstrate that the performance obtained is comparable to that observed when running the base systems on a comparable class of conventional EC2 VMs, thereby proving that our approach is a viable alternative to having to completely redesign either the engines or the serverless platforms.

2 MOTIVATION AND RELATED WORK

There is a growing amount of research on serverless computing [38]. Here we briefly focus on the work relevant to the point we are making in the paper: how to enable data analytics on serverless.

As pointed out, there are two approaches. One is to design new engines specific to serverless; the other is to develop more amenable serverless platforms. Of course, a combination of both approaches is also possible. Examples of the former approach are Lambada [30] and Starling [31]. Both propose query engines atop unmodified serverless platforms that use a varied number of techniques to work around the limitations of serverless platforms (mainly optimizing communication through storage and speeding up start-up times for many functions). These systems have inspired other work extending the ideas to, e.g., machine learning [26, 47]. Pixels-Turbo [21] augments a long-running VM-based data analytics system with specialized serverless workers to accelerate processing in response to sudden load spikes. Crackle [32] models a unified query engine that can execute both on top of VMs and unmodified serverless platforms to reduce overall cost.

 14th Annual Conference on Innovative Data Systems Research (CIDR '24). January 14-17, 2024, Chaminade, USA

Michael Wawrzoniak[1], Gianluca Moro[1], Rodrigo Bruno[2], Ana Klimovic[1], Gustavo Alonso[1]
[1]Systems Group, Dept. of Computer Science, ETH Zurich
[2]INESC-ID/Técnico, U. Lisboa

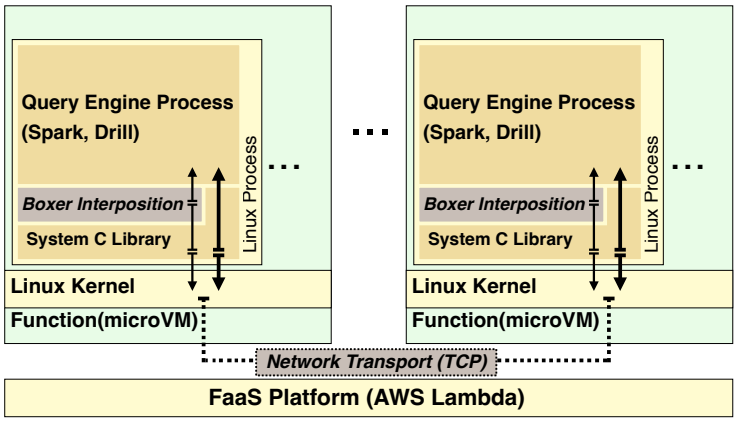

Figure 1: Unmodified distributed query engines running in parallel executing networked AWS Lambda functions.

Regarding alternative serverless platforms, the focus has been on providing, e.g., stateful function services [19, 40, 46] or extending serverless with, e.g., ephemeral storage solutions [27, 33] that offer a fast data plane option for serverless functions.

Neither of these approaches take advantage of existing data analytics systems (e.g., Apache Spark, Drill) and existing commercial serverless platforms (e.g., AWS Lambda). As a result, existing applications need to be rewritten to use the new platforms. It is also questionable whether the new serverless platforms will ever be as competitive and efficient as the native solutions of cloud providers.

The approach we put forward in this paper is based on a simple observation: serverless worker instances are already similar to small VMs (some are based on microVMs [18]). Supporting existing data analytics engines in serverless is, thus, a problem of ensuring that the environment available in serverless workers is sufficiently similar to the one available in serverful VM/container instances. With that, one should be able to use existing distributed data processing engines without changing either the underlying serverless platform nor the engines themselves.

3 DISTRIBUTED QUERY ENGINES IN SERVERLESS ENVIRONMENTS

The serverless FaaS paradigm is based on fine-grained, event triggered computations that are composed into dataflow graphs. Computations are based on Linux processes in both serverful VMs and serverless functions. From that perspective, applications such as distributed query engines running on VMs should, in principle, be able to run in serverless functions. Unfortunately, the environment and computation composition model of serverless platforms is significantly different from that of conventional VMs. Mainly, distribution in serverless platforms is based on an event-based dataflow composition model, while in conventional VM applications, it is based on parallel networked processes. Because of this, conventional distributed engines do not work on serverless platforms as they assume access to direct communication with other concurrently executing remote processes.

To bridge this gap, we leverage an evolution of Boxer [44] to transparently provide existing data engines with the required model of network of parallel executing processes on top of existing FaaS platforms. Doing this goes beyond just enabling networking as it has a number of consequences on the execution environment. Here we outline the most relevant aspects that need to be addressed.

3.1 Parallel function execution

Unmodified distributed query engines typically assume a static configuration of distributed processes continuously exchanging data. However, in serverless, function scheduling is based on events triggering function executions (control of available concurrency of the underlying resources is exposed in some commercial offerings [8]). This gives the FaaS platforms more freedom for scheduling while preserving the semantics of the dataflow computation, however, it does not match the environment that distributed query engines assume of a collection of active processes running in parallel.

To address this mismatch, we use an initialization mechanism that creates an engine-specific pool of function instances running in parallel. The pool of functions combines function types as needed (e.g., worker nodes, head nodes, auxiliary systems, etc.). Once the required combination of parallel functions is instantiated, the query engine processes are started in the function pool, matching the model that query engines expect.

3.2 Application transparency

To transparently expose the altered function environment to unmodified engines, a thin interposition layer between the query engine processes and the hosting functions (Figure 1) is introduced. As the query engine processes are started, the system dynamic linker is instructed to link a subset of system C Library function references to the interposition library (Figure 1). When the query engine processes issue these C Library function calls, they are first intercepted by the library and handled in a way that emulates the desired environment. For example, when a query engine process attempts to connect to a named process running in a remote function, instead of returning an error as it would happen within a serverless function, the name resolution and connect calls are intercepted, network addresses are provided and a connection setup is performed (see below), returning a valid stream socket connected to

the named remote function. The query engine process is unaware of the additional connection setup performed and proceeds just as if it would have connected to a remote host process. The interposition layer introduces no data plane performance overhead; no data plane functions are intercepted, e.g., once a connection is established, all data writes and reads are processed without the interposition layer being involved.

In addition to the transparency from the perspective of the individual query engine processes, we aim to provide a transparent system orchestration environment. Query engines are composed of sub-services (e.g, head nodes, worker nodes, meta-data services, coordination services, etc.) that are commonly managed using container orchestration tools, such as Kubernetes [3], Docker Swarm [2], or Docker Compose [11]. To allow for orchestration-level transparency, query engine container images can be derived from Boxer base images that can be run via native container runtimes or can execute as FaaS instances. In this paper, all experiments were orchestrated using Docker Compose that either started containers using container runtimes on EC2 VMs (baseline) or as AWS Lambda instances (unmodified engines on serverless).

3.3 Network transport

In order for the query engine processes executing in different functions to communicate using the expected socket interface, a network transport mechanism must be provided. Most query engines rely on stream socket semantics to communicate, commonly implemented by TCP protocol (datagram semantics are not used by any of the query engines that we inspected.) Unfortunately, commercially available serverless FaaS platforms do not provide a mechanism to establish TCP connections where the endpoints are different function instances. Functions can establish TCP connections to destinations outside of the FaaS service but not to other functions. To address this limitation, stream-oriented network transport between processes running in different functions is provided. The primary method is using TCP connections that are established using NAT hole-punching techniques. Transport based on IP-forwarding through EC2 instances is also available[1], however, currently, the NAT hole-punching transport is preferable from a performance perspective and is used for the stream sockets used by the query engines in this work.

4 EVALUATION

We demonstrate the feasibility of the proposed approach and present initial measurements of two unmodified existing distributed query engines, Apache Spark and Apache Drill, running on a commercially available serverless FaaS platform, AWS Lambda. We report comparative TPC-H benchmark measurements for the two query engines executing in AWS Lambda functions and AWS EC2 virtual machines showing that distributed query execution performance in serverless functions is comparable to that of a class of EC2 instances. We then examine the overhead of the per-query engine initialization times and propose future improvements.

Both Apache Spark and Apache Drill support large-scale analytics with distributed query execution. In the conventional mode of operation, the systems are run on long-running dedicated clusters or virtual machines. In contrast to that environment, we intend to use these systems for serverless data analytics. Therefore we run these engines using the evolution of Boxer (Section 2) in short-lived networked AWS Lambda serverless functions. We instantiate Apache Spark and Apache Drill instances when there is a query to be run and shut down immediately after producing results.

[1]Other stream transports can be implemented, e.g., ranging from S3 storage to QUIC network protocol.

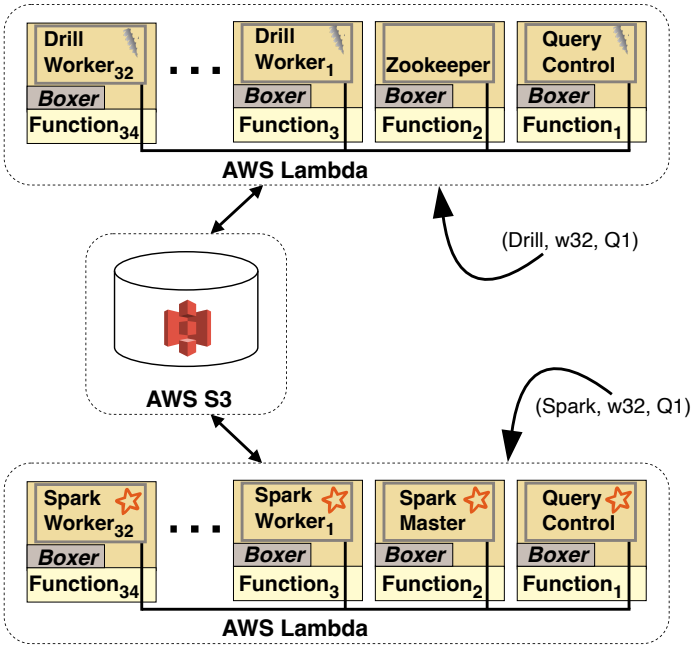

Figure 2: Experimental setup: Example of (bottom) Apache Spark (32 workers, master, and query control node) in AWS Lambda instantiated to execute query Q1 on data stored in AWS S3. Apache Drill (top) (32 workers, Apache Zookeeper instance, query control node) instantiated in AWS Lambda to execute query Q1 on AWS S3.

4.1 Experimental setup

The experiments presented here use the TPC-H dataset at scale factors(SF) 10, 30 and 100 with data stored in Parquet format, compressed with Snappy, and split into 100MB partitions stored on AWS S3. The resulting compressed dataset sizes range from 3GB for SF-10 to 32GB for SF-100, with the largest relation of SF-100 containing 600 million rows.

The AWS Lambda measurements are based on functions with 10GB of memory and 6 virtual cores, the largest AWS Lambda functions currently available. All query engine nodes and auxiliary functions (explained below) execute in their own AWS Lambda functions (Figure 2). In the Apache Spark experiment, there are 8, 16, or 32 worker nodes and a master node. In the case of Apache Drill, there are 8, 16, or 32 worker nodes and a single-node Apache Zookeeper [24] instance (also instantiated in a dedicated function) that Drill relies on for coordination. In addition to the above nodes, every query engine instantiation includes a query control function

Michael Wawrzoniak[1], Gianluca Moro[1], Rodrigo Bruno[2], Ana Klimovic[1], Gustavo Alonso[1]
[1]Systems Group, Dept. of Computer Science, ETH Zurich
[2]INESC-ID/Técnico, U. Lisboa

Figure 3: TPC-H (scale factor 100) query execution times for unmodified Apache Spark and Apache Drill running in AWS Lambda and AWS EC2. Median times of 3 executions of each query for each configuration, error bars are min. and max. times.

that is responsible for waiting for the query engine to be ready, submitting the specified query, and waiting for the result.

For each TPC-H query measured, we instantiate all of the above-described query engine instance functions, run a single query, and terminate all of the functions. Each TPC-H query configuration is repeated 3 times. In this experiment, we want to factor out the effects of using warm functions and caching. To ensure that each query is started in fresh cold functions, after each query, we reset the function registrations so that the platform considers them as new functions, guaranteeing cold starts.

To relate the measurements in AWS Lambda to the virtual machine environment, we performed the AWS EC2 measurements in a symmetric way. Instead of AWS Lambda functions, AWS EC2 virtual machines are instantiated. For each query execution we instantiate a new set of EC2 virtual machines, execute a single query and delete the virtual machines. All measurements are based on t2.2xlarge EC2 instances configured in unlimited mode (to avoid possible CPU throttling) and limited by the Linux kernel to 6 virtual cores and 10GB of memory (from 8 virtual cores and 32GB of memory). We found that virtual machines configured this way provide similar performance to the AWS Lambda functions that we use (also with 6 virtual cores and 10GB of memory).

All of the described nodes are configured as container images derived from Boxer base images. Containers derived from the Boxer base images can be used by conventional container orchestration tools, or they can be selectively run as AWS Lambda functions. We ran all of the benchmarks using the same container images on EC2 instances as AWS Lambda and used Docker Compose [11] as the container orchestrator for both.

All of the measurements are based on Spark version 3.32, Drill version 1.21.1, Zookeeper 3.7.1, and Amazon Corretto 17 JVM package. The systems are not performance-tuned, the measurements represent close to out-of-box unoptimized configurations. All measurements were performed on AWS eu-central-1 region using x86_64 CPU architectures.

4.2 TPC-H measurements

Apache Spark and Apache Drill per-query TPC-H measurements on AWS EC2 and AWS Lambda are shown in Figure 3 and Figure 4. The main observations from this experiment are that

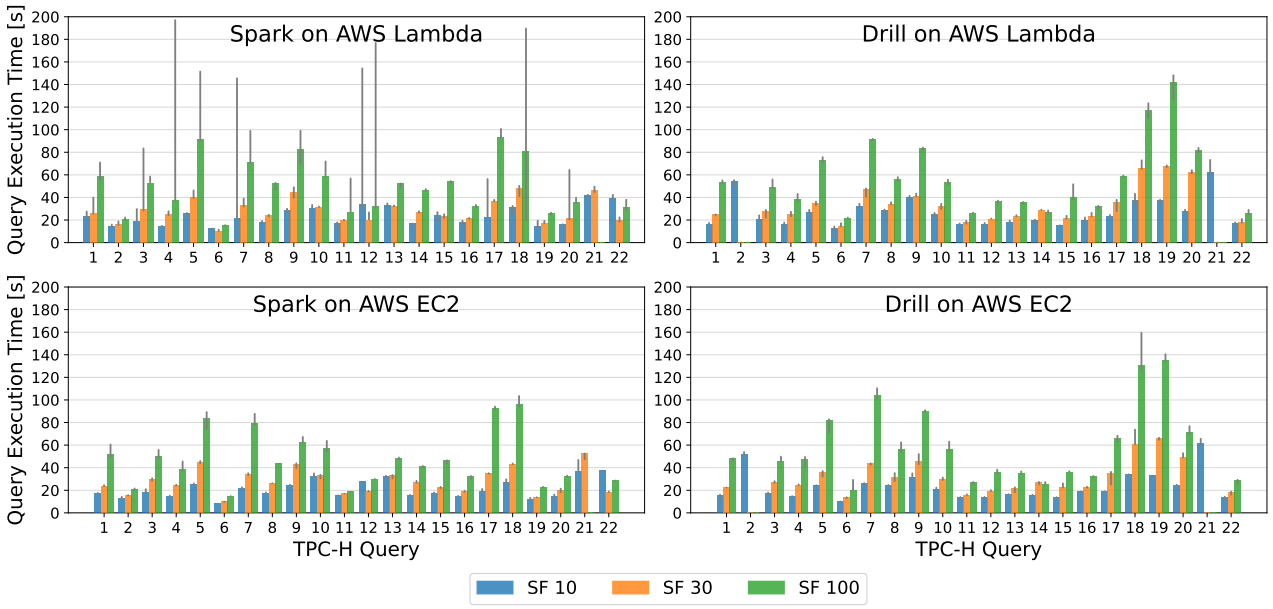

Figure 4: TPC-H query execution times for different scale factors (SF) for unmodified Apache Spark and Apache Drill running in AWS Lambda and AWS EC2 with 8 workers. Median times of 3 executions of each query for each configuration, error bars are min. and max. times.

(1.) *Unmodified Apache Spark and Apache Drill distributed query engines can run in AWS Lambda serverless functions,*
(2.) *the distributed query execution performance in AWS Lambda is comparable to using a class of networked EC2 virtual machines.*

Figure 3 shows observed query execution times for TPC-H SF-100 for different number of worker nodes (8,16,32), and Figure 4 shows query execution times for different scale factors (SF-10, SF-30, SF-100) all using 8 worker nodes. Comparing the measurements based on VMs and functions we make the following observations:

Completeness: First, except for a single case (described below), the same sets of queries complete successfully on EC2 VMs as on Lambda functions.

Spark completed all queries on AWS Lambda and EC2 in the SF-10 and SF-30 measurements and SF-100 with 32 worker nodes. However, other SF-100 configurations with fewer than 32 workers did not complete all queries. This is because when Spark was configured with only 8 or 16 worker nodes for SF-100, query 21 could not complete due to the workers running out of memory. The same issue was observed on both platforms.

Apache Drill, in all shown configurations except for SF-10, was unable to complete query 2 due to the workers running out of memory. The same reason prevented Drill from completing query 21 using 8 workers for SF-30 and SF-100. The same issues were observed on both platforms. However, in the case of SF-100 in 16 and 32 worker configurations, Drill was able to complete query 21 on EC2 but not on Lambda. This difference is due to AWS Lambda functions having a limited number of file descriptors available (1024 in total.) For these two configurations, Apache Drill workers attempted to create a sufficiently large number of temporary files that the file descriptor limit was reached and execution terminated. We did not find a Drill configuration option to reduce this unusually large number of temporary files. This is the only case we observed where the query execution was not possible using AWS Lambda functions but was possible using AWS EC2 VMs with similar resources.

Dispersion: Second, depending on the data processing system, query execution times can show a larger degree of dispersion in Lambda compared to EC2.

There is a significantly larger degree of dispersion in measurements for Spark running in Lambda compared to that in EC2. However, this difference is not shared by both systems. For Drill, the observed dispersion levels are similar in Lambda and EC2.

The dispersion level for Spark increases with the number of workers used, the number of queries where the maximum execution times is greater than 2× the median execution time is 4, 7 and 11 for 8, 16, and 32 workers configurations respectively (Figure 3). The dispersion level does not show a significant dependence on the scale factor for measurements with 8 workers (Figure 4). We have not yet identified the mechanism that produces this increased dispersion that is specific to Spark on Lambda, however, it is worth observing that even in the cases when the difference in median times between Spark in EC2 and Lambda is significant, the minimum time in Lambda approaches that of EC2, suggesting that if the high variance can be addressed, the query execution times will consistently approach those of EC2.

Performance: The query execution times of Spark and Drill vary but overall are comparable in AWS Lambda functions and EC2 VMs.

Michael Wawrzoniak[1], Gianluca Moro[1], Rodrigo Bruno[2], Ana Klimovic[1], Gustavo Alonso[1]
[1]Systems Group, Dept. of Computer Science, ETH Zurich
[2]INESC-ID/Técnico, U. Lisboa

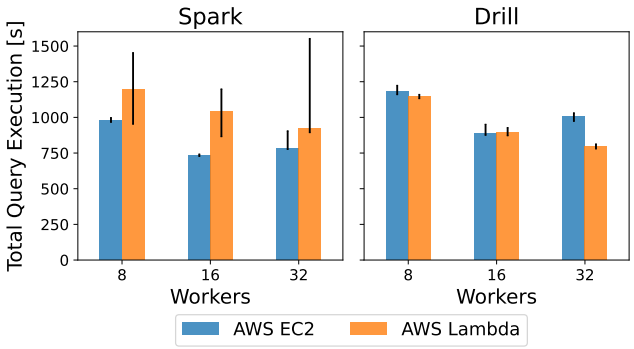

Figure 5: Total time to run all queries (excludes Q2 and Q21) for SF-100.

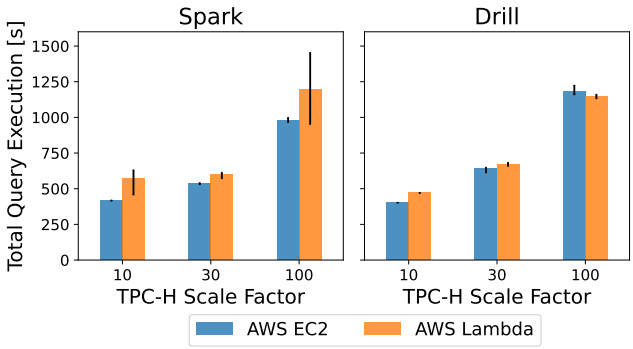

Figure 6: Total time to run all TPC-H queries (excluding Q2 and Q21) using 8 worker nodes.

Eng.	Workers	Plat.	Median	Mean	Min	Max
Spark	8	EC2	978.76	980.91	965.91	998.08
Spark	8	Lambda	1195.08	1200.31	952.53	1453.30
Spark	16	EC2	731.26	733.52	726.85	742.45
Spark	16	Lambda	1039.12	1034.42	865.39	1198.75
Spark	32	EC2	780.48	820.46	775.76	905.14
Spark	32	Lambda	922.02	1122.65	893.66	1552.27
Drill	8	EC2	1178.38	1187.34	1159.89	1223.75
Drill	8	Lambda	1149.52	1147.24	1131.93	1160.28
Drill	16	EC2	886.40	904.11	874.92	950.99
Drill	16	Lambda	891.85	897.18	871.70	927.99
Drill	32	EC2	1007.71	1003.96	972.75	1031.41
Drill	32	Lambda	794.73	795.77	779.20	813.37

Table 1: Total time to run all TPC-H queries (excludes Q2 and Q21) for SF-100.

SF	Workers	Spark EC2	Spark Lambda	Spark L.%	Drill EC2	Drill Lambda	Drill L.%
SF-10	8	2.35	2.10	0.89	2.94	2.44	0.83
SF-30	8	1.83	1.99	1.09	1.83	1.72	0.94
SF-100	*8*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
SF-100	16	1.34	1.15	0.86	1.33	1.29	0.97
SF-100	32	1.25	1.30	1.03	1.17	1.45	1.24

Table 2: Speedups relative to SF-100 with 8 workers to run all TPC-H queries as scale factor is decreased (to SF-30 and SF-10) or as the number of workers is increased (to 16 and 32). L.% is the fraction of the EC2 speedup achieved by Lambda.

The sums of the query execution times for all TPC-H queries at SF-100 are reported in Table 1 and Figure 5. Times for each of the 3 executions are summed over all queries, except for queries 2 and 21, which are excluded from the summations because not all configurations completed these two queries. The median of the sums for Spark in AWS EC2 is always faster than in Lambda (between 1.2× and 1.4×), where the high-variance executions contributed significantly to the aggregate differences. If only the aggregate of the minimum times were considered (Figure 3), the difference would be significantly reduced, showing that if the variance in Spark executions can be addressed, the absolute times between EC2 and Lambda would be even closer. Apache Drill, exhibiting much less variance, shows the EC2 and Lambda median of sum times to be much closer; for 16 workers configuration, the difference is minimal (under 1.01×) in favor of EC2, and in the case of 8 and 32 workers, the median of the sum times for Drill is faster in Lambda than in EC2 configuration, with 1.02× and 1.26× speedup respectively.

Scaling: Relative scaling properties of Spark and Drill are similar in AWS Lambda and AWS EC2. Scaling query engines by the number of worker nodes (Figure 5) or by the data set scale factor (Figure 6) does not result in degenerate scaling properties that could be observed if, for example, the network bandwidth of the Lambda functions was shared.

Table 2 lists relative speedups achieved by decreasing the scale factor or increasing the number of workers. The speedups for Spark and Drill on both EC2 and Lambda platforms are relative to query execution times of all queries (except Q2 and Q21) for SF-100 and using 8-worker nodes. The top two rows show the speedup achieved as the scale factor is decreased to SF-30 and SF-10, and the bottom two rows show the speedups observed as the number of workers is increased to 16 and 32 (e.g. the speedup achieved by Spark on Lambda by decreasing the scale factor from SF-100 to SF-30 while keeping the number of workers constant at 8 is 1.99). The fraction of the speedup achieved by Spark on Lambda relative to EC2 ranges from 0.86 to 1.09, and for Drill from 0.83 to 1.24.

The difference in throughput between Lambdas and the EC2 instances we selected can be a factor in these scaling (and absolute times) differences. The throughput between the EC2 instances used is 975Mbit/s (median), and between the Lambda instances, it is 629Mbit/s (median,) both with low variance over time. However, in the first 5 seconds of Lambda execution, the temporary throughput can be observed as high as 2.8Gbit/s. As the number of Lambdas increases, for some of the queries, this temporary burst may help to accelerate a larger fraction of the execution, in other cases, the higher steady-state throughput may be advantageous.

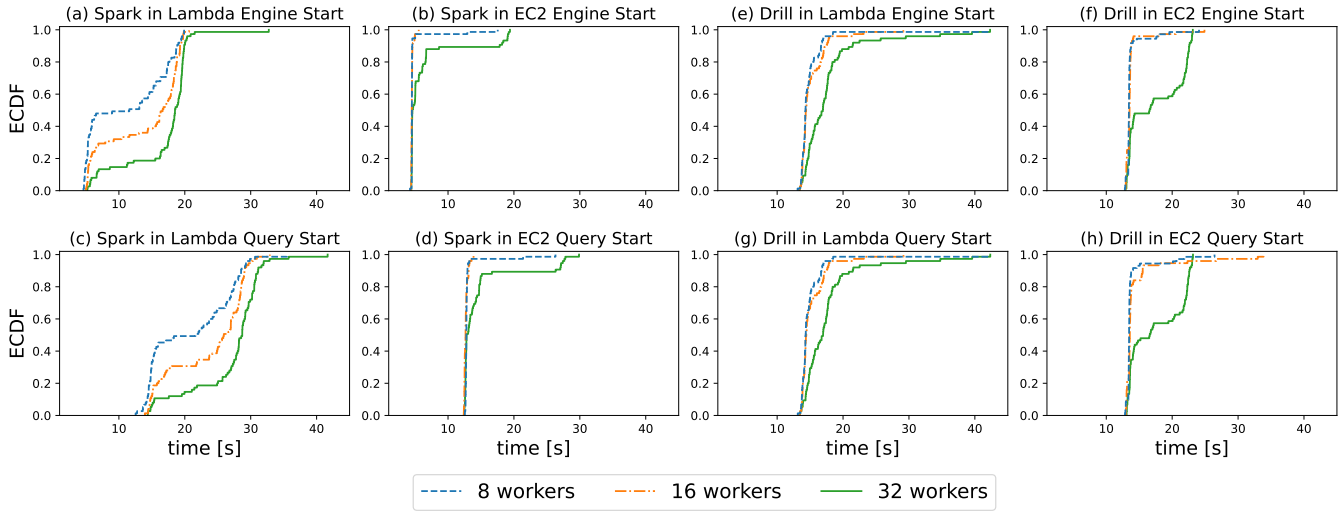

Figure 7: Engine initialization times (details in Section 4.3).

Engine	Workers	Median	Mean	Std Dev	Min	Max
Spark	8	16.46s	20.25s	6.16s	12.54s	35.47s
Spark	16	26.99s	24.50s	5.25s	13.90s	32.90s
Spark	32	28.27s	26.19s	5.32s	14.14s	32.33s
Drill	8	14.28s	14.54s	1.53s	13.01s	25.62s
Drill	16	14.53s	15.46s	2.61s	13.46s	29.17s
Drill	32	17.70s	18.77s	5.40s	13.40s	42.25s

Table 3: Time to submit a query in Apache Spark and Apache Drill in AWS Lambda. Corresponds to Figure 7(c) and 7(g).

4.3 Query engine initialization

Executing queries in serverless functions involves additional query engine startup as the engines are instantiated. In the limit, a new query engine instance can be started to run each query, or an engine may need to be restarted if the function duration limit is reached (currently, AWS Lambda functions have 15 minutes duration limit). We measure the time required to start Apache Spark and Apache Drill instances from the moment the requested set of AWS Lambda functions is available to when a query can be submitted for execution (Table 3).

We subdivide the query engine initialization time into two stages: the time to start the engine and the time to start processing queries ('Engine Start' and 'Query Start' respectively in Figure 7). The time to start the engine ('Engine Start') includes the time for the master node (in Spark) or Zookeeper (in Drill) to start and for all the worker nodes to start and join. At this point, we consider the query engine started but not yet ready to begin query execution. In the next stage ('Query Start'), each engine needs to be appropriately initialized so that it can execute the query. In the case of Drill, the table location catalog must be updated to reflect the TPC-H tables stored in S3, and only then the query is submitted for execution. In the case of Spark, when the cluster is started, it is ready to allocate executors on its workers for Spark applications. Depending on the Spark API used, different initialization procedures can follow; in our experiments, we allocated a single executor per worker and used Spark SQL Scala API to configure the TPC-H tables and submit the query.

As expected, as the number of workers increases, so does the time to start the engine and begin the query execution. In the case of Apache Drill in AWS Lambda, the median times to start query execution range from 14.28*s* to 17.70*s* for 8 to 32 workers (Table 3). However, the observed maximum times are significantly higher, up to 42.25*s*. The distribution of these times, Figure 7(g), shows that only a small fraction of the Drill initialization times have such high start times, and we see from Figure 7(e) that the long tail propagates from the time needed for workers to start. One possible source of this is the system waiting for straggler workers to join. Since this delay is before the query execution starts, one possible solution is to start more workers than required. Once the required number of workers joins, the remaining slower workers can be released.

In the case of Apache Spark in AWS Lambda, median times to start queries also increase with worker count, from 16.46*s* to 28.99*s* for 8 to 32 workers. However, unlike in Apache Drill, we see a higher degree of dispersion Figure 7(c). We also observe that this is most likely contributed by the time required for the workers to join Figure 7(a). Based on comparison with the same measurement in EC2 Figure 7(b), we see that this level of dispersion is specific to AWS Lambda, as Spark engine start times in EC2 show significantly lower dispersion. The above-mentioned approach of starting a small number of extra workers may be less appropriate here.

A promising and more general solution to the overhead of the query engine startup time may be to use function snapshots. AWS Lambda already provides a method of starting functions from previously taken function snapshots [12]. We believe this could be adapted to reduce the query engine initialization times. To execute a query, functions would be configured to start from an already initialized query engine that is ready for query execution.

Michael Wawrzoniak[1], Gianluca Moro[1], Rodrigo Bruno[2], Ana Klimovic[1], Gustavo Alonso[1]
[1]Systems Group, Dept. of Computer Science, ETH Zurich
[2]INESC-ID/Técnico, U. Lisboa

5 DISCUSSION

In this section we consider the reported results from the perspective of how the cloud has evolved over the years, discuss the limitations of the current system, and briefly explore promising research directions it opens up.

5.1 Historical Perspective

Functions in today's FaaS platforms are considered lightweight and resource-restricted. Relative to the available VMs, they are small; the largest available AWS Lambda today has only 10GB of memory and 6 vCPUs. From this perspective, running off-the-shelf data analytics platforms that are designed to run natively on much larger VMs may seem like a mismatch. However, it is worth considering that when some of the most popular datacenter systems used today were originally designed and developed, their target environment was closer to today's AWS Lambda functions than to today's AWS EC2 VMs. For example, Zookeeper became an Apache Software Foundation project in 2008 [13] and the Spark project started just a year later in 2009 [7]. At that time, newly announced AWS EC2 High-CPU instance types [4], c1.medium had 1.7GB of memory and 2 vCPUs, and c1.xlarge had 7GB of memory and 8 vCPUs. When Zookeeper was commonly used, and just before the Spark project started, the AWS EC2 'High-CPU Extra Large Instance' VM had comparable CPU and memory resources to today's AWS Lambda function.

As these figures show, hardware can evolve faster than software, something that is especially true in the cloud. While the available VMs and supporting platforms have significantly grown, the underlying system architecture of data analytic engines and related services (e.g., Zookeeper, Spark) has remained largely unchanged. This seems to indicate that, possibly with some amount of re-configuration and tuning, data analytics engines can be a good match for contemporary *networked* serverless environments. As we will discuss later, the combination leads to interesting ideas when such engines can be quickly instantiated in the type of short-lived data center provided by serverless functions.

5.2 Limitations

As we have shown in Section 4.3, existing data analytics platforms (together with the underlying JVM runtime) can have long initialization times. These initialization times might seem to negate the benefits of using elastic serverless platforms. However, since this is a problem that is shared by many other serverless applications, serverless platforms now offer functionality to accelerate initialization times by starting functions from previously taken snapshots [12]. We plan to take advantage of this technique to accelerate query engine initialization times. For this, we need to investigate how to leverage the AWS Lambda snapshots to snapshot and restore collections of networked functions. This will raise a number of questions. First, what to snapshot - how specific do the engine configurations have to be when they are snapshotted, e.g., do we need to create different snapshots for engines with different worker counts, or can we dynamically select the number of workers to be restored. Second, at what point in the query engine initialization phase do we need to take the snapshots, e.g., to make Spark available to all types of applications we should snapshot at the engine start time. However, from the difference in between Figure 7(a) and (c), we see that to use Spark SQL interface we would still need to pay overhead for the allocation of the appropriate executors to submit the query. We could create snapshots for different sets of executor instantiations, so in the case of Spark SQL we can submit the query right after restoring the snapshots. These questions need to be studied to successfully leverage the already available snapshotting technology to reduce the initialization overhead.

Additionally, there are limitations that have an impact on which use cases are currently a good fit for serverless. Serverless platforms currently impose a fixed time limit (e.g., AWS Lambda function invocations cannot take longer than 15 minutes) that restricts serverless use cases to focus on fast-executing queries. Furthermore, serverless workers usually are not as flexible compared to VMs with regard to resource bundles as providers typically offer a small set of resource configurations (e.g., AWS Lambda only allows users to select the amount of memory, CPU is allocated proportionally). The current pricing model also narrows down the workload patterns that are cost efficient compared to long-running engines [30, 31].

The prototype used in this study has limitations that we continue to address. We are investigating some of the failures we experienced during the experiments to determine if they are due to errors in the implementation or if there are additional mechanisms that we need to implement to improve robustness and scalability. The system C Library interposition layer does not yet cover sufficient interface surface area to support all applications. As we experiment with more query engines, we will expand the support, improve robustness, and cover more of the corner cases. We believe to be close to supporting a wider range of engines, in particular, we have been experimenting with Trino [16], Databend [10], and Clickhouse [9], which will soon become ready to run on serverless like the systems demonstrated in the paper.

5.3 Opportunities

The proposed approach can be extended to other settings beyond just running individual (or sets of) queries. For long-running query engines, sudden bursts of query volume can quickly lead to congestion and, as a result, long response times. We are currently studying how to use serverless workers as an extension of off-the-shelf serverful query engine deployments to quickly accommodate rapid workload fluctuations. In this scenario, serverless networked functions are used to deploy additional worker instances of a query engine that are used only for the duration of the workload spike. By using serverless functions to accommodate such workload bursts, serverful infrastructure overprovisioning can be significantly reduced. Recently, Pixels-Turbo [21] system described a custom system where serverless workers can be instantiated to execute subplans to accelerate query execution of VM-based system. However, in contrast to our approach, specialized serverless query executors had to be implemented and integrated with Pixels [20] system, and not taking advantage of function-to-function networking. This type of rapid scaling into networked serverless functions is not specific to query engines, other types of data processing systems (and beyond) can benefit from this paradigm. For example, stream

processing systems such as Apache Flink [6] can reduce their level of overprovisioning if they can temporarily scale out to networked Lambda functions to quickly absorb load spikes while handling steady-state load using the more cost-effective VMs.

The ability to instantiate off-the-shelf data analytics systems in serverless also has the potential to help take advantage of data locality by facilitating executing (sub)queries closer to the data. Long-running VM-based systems could not afford the flexibility to be fully or partially instantiated close to the queried data on per-query granularity. Using Boxer, a long-running VM-based system may be transparently augmented by short-lived serverless workers instantiated close to a remote data source as queries that reference it arrive. Depending on the workloads, this may mean instantiating enough new workers to fully execute the query, or just enough to perform appropriate data reduction close to the data source. This can be especially beneficial if the workload is composed of queries that reference diverse data sources that are close to FaaS platforms, such as in different regions of a cloud provider or even across different cloud providers. We have not yet experimented with using Boxer as a wide-area overlay or across different providers, but we plan to explore this direction, possibly taking advantage of some recent cost, latency, and throughput optimization studies [25, 35].

The ability to run existing query engines on serverless gives rise to a new question - how to choose the right engine and configuration to execute a given query? Factors like expected execution speed, resource budget, engine configuration options, or feature sets are just a few of the dimensions that can now be considered at the per-query granularity, leading to exciting new research opportunities. As we increase the set of supported query engines, we are beginning to consider this line of research [43].

Though we advocate for running off-the-shelf applications on existing commercial serverless platforms, such as AWS Lambda, it is also important to consider how serverless platforms can and should evolve to optimize performance and resource efficiency. For example, Dandelion [28] proposes a declarative serverlesss programming model that strictly separates untrusted user compute functions and platform-provided I/O functions. Dandelion's programming model exposes dataflow to the platform, enabling optimizations like pre-fetching from storage, locality-aware scheduling, and efficient inter-function communication. We plan to explore how a serverless platform like Dandelion would enable building data processing engines directly on top of the declarative function execution model interface instead of through an overlay like Boxer. However, it is also worth exploring how to continue providing the familiar abstractions that traditional data analytics engines expect while evolving the underlying platform. Boxer may provide a path to expose the advantages of platforms such as Dandelion to systems such as data analytics engines that expect the network-of-hosts programming models.

Depending on the workload, it can be beneficial to augment the serverless environment with caching. A number of specialized serverless caching solutions have been proposed [17, 29, 39], some being able to leverage publicly available systems serverless platforms [34, 41]. Instead of relying on serverless-specific solutions that work around serverless limitations, we can use unmodified of-the-shelf distributed caching systems such as Memcached [14] or Redis [15] to be instantiated alongside data analytics platforms in our networked serverless functions environment. Beyond improving single-system performance, especially when multiple queries fetch common remote data, such caches can be leveraged to reduce data movement when chaining executions of multiple different systems together in serverless functions. For instance, when one system is used for a preprocessing step, followed by data analytics queries based on a different system, and then machine learning tasks requiring yet another system, all leveraging the caching systems running in the networked serverless functions for passing the intermediate state between the stages. Such pipelines could be composed of a combination of specialized serverless systems and off-the-shelf systems running in dynamic sets of networked functions.

Finally, serverless functions have more limited resources compared to currently available virtual machine options. They can also be scaled down to very small execution environments (AWS Lambda as low as 128MB of memory and a fractional vCPUs) that, for some workloads, might be the right size. Given that the existing engines target virtual machine environments, there may be an interesting opportunity to consider supplementing the existing engine workers with variants that are meant to be short-lived, lighter weight, possibly more limited in functionality, and more specialized to their target functions and the resource-constrained execution environments of serverless functions. Introducing these specialized engine 'worklets', initially ranging from specialized configurations of full workers to eventually even single-operator micro-workers, is a promising trajectory on the path to minimize further the gap between the existing query engines and the flexibility of serverless.

6 CONCLUSION

This paper proposes a new direction for serverless data analytics — using existing unmodified off-the-shelf data analytics engines on existing unmodified serverless infrastructure. We validate the feasibility of the approach by transparently running two such distributed engines (Apache Spark and Apache Drill) on a commercially available serverless platform (AWS Lambda). Our initial results demonstrate that these serverless deployments have comparable performance to those based on a class of networked VMs, charting a new way to bridge the gap between existing distributed query engines and serverless.

REFERENCES

[1] 2020. AWS Lambda. Retrieved 2020-08-17 from https://aws.amazon.com/lambda
[2] 2022. Docker Swarm overview. Retrieved 2022-04-15 from https://docs.docker.com/engine/swarm/
[3] 2022. Kubernetes. Retrieved 2022-04-15 from https://kubernetes.io/
[4] 2023. Amazon EC2 now provides High-CPU instance types. Retrieved 2023-12-03 from https://aws.amazon.com/articles/feature-guide-amazon-ec2-high-cpu-instance-types/
[5] 2023. Apache Drill. Retrieved 2022-10-20 from https://drill.apache.org/
[6] 2023. Apache Flink. Retrieved 2023-03-01 from https://flink.apache.org/
[7] 2023. Apache Spark history. Retrieved 2023-12-03 from https://spark.apache.org/history.html
[8] 2023. AWS Lambda provisioned concurrency. Retrieved 2023-07-27 from https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html
[9] 2023. ClickHouse. Retrieved 2023-06-20 from https://clickhouse.com/
[10] 2023. Databend. Retrieved 2023-06-20 from https://databend.rs/
[11] 2023. Docker Compose. Retrieved 2023-07-27 from https://docs.docker.com/compose/
[12] 2023. Improving startup performance with Lambda SnapStart. Retrieved 2023-03-01 from https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html

[13] 2023. Initial ZooKeeper code contribution from Yahoo! Retrieved 2023-12-03 from https://issues.apache.org/jira/browse/ZOOKEEPER-1
[14] 2023. Memcached. Retrieved 2023-12-03 from https://memcached.org/
[15] 2023. Redis. Retrieved 2023-12-03 from https://redis.io/
[16] 2023. Trino. Retrieved 2023-06-20 from https://trino.io/
[17] Mania Abdi, Sam Ginzburg, Charles Lin, Jose M Faleiro, Íñigo Goiri, Gohar Irfan Chaudhry, Ricardo Bianchini, Daniel S. Berger, and Rodrigo Fonseca. 2023. Palette Load Balancing: Locality Hints for Serverless Functions. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. ACM.
[18] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*.
[19] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful Functions as a Service in Action. *Proc. VLDB Endow.* 12, 12 (2019), 1890–1893.
[20] Haoqiong Bian and Anastasia Ailamaki. 2022. Pixels: An Efficient Column Store for Cloud Data Lakes. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 3078–3090.
[21] Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. 2023. Using Cloud Functions as Accelerator for Elastic Data Analytics. *Proc. ACM Manag. Data* 1, 2, Article 161 (jun 2023), 27 pages.
[22] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Distributed Transactions on Serverless Stateful Functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*. 31–42.
[23] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR*.
[24] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems *(USENIX ATC'10)*.
[25] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. 2023. Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1375–1389.
[26] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data*. 857–871.
[27] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI*. 427–444.
[28] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function *(SoCC '23)*. 81–92. https://doi.org/10.1145/3620678.3624648
[29] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: An Opportunistic Caching System for FaaS Platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244. https://doi.org/10.1145/3447786.3456239
[30] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD*.
[31] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD*.
[32] Matthew Perron, Raul Castro Fernandez, Michael Cafarella, and Samuel Madden. 2023. Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools. In *SIGMOD*.
[33] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *NSDI 19*.
[34] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 122–137.
[35] Noga H. Rotman, Yaniv Ben-Itzhak, Aran Bergman, Israel Cidon, Igor Golikov, Alex Markuze, and Eyal Zohar. 2022. CloudCast: Characterizing Public Clouds Connectivity. *CoRR* abs/2201.06989 (2022). arXiv:2201.06989 https://arxiv.org/abs/2201.06989
[36] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* 64, 5 (April 2021), 76–84.
[37] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* 64, 5 (April 2021), 76–84.
[38] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Comput. Surv.* 54, 11s, Article 239 (nov 2022), 32 pages.
[39] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *PVLDB* (2020).
[40] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (2020), 2438–2452.
[41] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *USENIX FAST*.
[42] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Annual Technical Conference (USENIX ATC 18)*. Boston, MA, 133–146.
[43] Michael Wawrzoniak, Rodrigo Bruno, Ana Klimovic, and Gustavo Alonso. 2023. Ephemeral Per-query Engines for Serverless Analytics. In *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (VLDB 2023), Vancouver, Canada, August 28 - September 1, 2023 (CEUR Workshop Proceedings, Vol. 3462)*.
[44] Michal Wawrzoniak, Ingo Müller, Rodrigo Bruno, and Gustavo Alonso. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *CIDR*.
[45] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2018. Anna: A KVS for Any Scale. In *ICDE*.
[46] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 83–97.
[47] Yuncheng Wu, Tien Tuan Anh Dinh, Guoyu Hu, Meihui Zhang, Yeow Meng Chee, and Beng Chin Ooi. 2022. Serverless Data Science - Are We There Yet? A Case Study of Model Serving. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*.
[48] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65.