Journal of Parallel and Distributed Computing 183 (2024) 104763

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

A Seer knows best: Auto-tuned object storage shuffling for serverless analytics

Germán T. Eizaguirre, Marc Sánchez-Artigas *

Computer Science and Maths, Universitat Rovira i Virgili, Spain

ARTICLE INFO

Article history:
Received 31 October 2022
Received in revised form 11 July 2023
Accepted 30 August 2023
Available online 7 September 2023

Keywords:
Serverless computing
Shuffle
I/O optimization
Object storage

ABSTRACT

Serverless platforms offer high resource elasticity and pay-as-you-go billing, making them a compelling choice for data analytics. To craft a "pure" serverless solution, the common practice is to transfer intermediate data between serverless functions via serverless object storage (IBM COS; AWS S3). However, prior works have led to inconclusive results about the performance of object storage systems, since they have left large margin for optimization. To verify that object storage has been underrated, we devise a novel shuffle manager for serverless data analytics called Seer. Specifically, Seer dynamically chooses between two shuffle algorithms to maximize performance. The algorithm choice is made online based on some predictive models, and very importantly, without end users having to specify intermediate shuffle data sizes at the time of the job submission. We integrate Seer with PyWren-IBM [31], a well-known serverless analytics framework, and evaluate it against both serverful (e.g., Spark) and serverless systems (e.g., Google BigQuery, Caerus [46] and SONIC [22]). Our results certify that our new shuffle manager can deliver performance improvements over them.

1. Introduction

In the pursuit of higher elasticity, major cloud providers have recently begun to offer new computation infrastructures such as serverless computing. While initially geared towards web microservices and IoT applications, the latest literature has started to scrutinize the role of serverless computing in data-intensive applications [16,11,38,6,22,46,48,2]. Altogether, this handful of studies has shown that serverless computing is far an appealing choice for data analytics.

Typically, data analytics frameworks use a directed acyclic graph (DAG) to represent the computation logic of an analytics job, with stages as its vertices, and the dependencies between stages as its edges. Each stage consists of a set of parallel tasks, each processing a partition of the dataset. While traditional server-centric deployments utilize clusters provisioned with a fixed pool of compute resources to run the parallel tasks in a job, serverless deployments execute them as cloud functions [11,38,22,46]. Billed only for their running time (pay-as-you-go model), along with their fast provisioning times, functions enable the allocation of compute resources at the task level, avoiding the under- and over-provisioning of resources that occur when they are provisioned at the job granularity. For instance, the intermediate data size across various stages in a typical TPC-DS query [28] ranges from 0.8 MB to 66 GB, a difference of 5 orders of magnitude! [19].

The problem of serverless shuffle. Despite these benefits, directly using a serverless platform for processing a DAG is problematic. By design, serverless functions are not **network addressable**, so direct point-to-point communication is hard. This limitation immediately reverberates into data shuffles, which naturally turn up in many data transformations like `groupBy` or `join` operations. But, what is a data shuffle? In a nutshell, **shuffling** is the process of redistributing or re-partitioning the data at the different machines across new partitions. To better understand this, pretend that a big table with two columns, e.g., `Date` and `Customer`, is split by rows, with different ranges of rows stashed on two different partitions as depicted in Fig. 1. If a user wished to group the data by the column `Customer`, the execution of the `groupBy` operation would require to redistribute the data in each worker into new partitions as illustrated in Fig. 1. This repartitioning of data is what is called a shuffle. In this specific case, the redistribution of data between the two partitions is performed using a hash partitioner, i.e., which decides the output partition based on the hash code computed for the column `Customer`.

What is the main problem? The biggest issue is that shuffles require `all-to-all` communication between all workers, which can-

* Corresponding author.
E-mail addresses: germantelmo.eizaguirre@urv.cat (G.T. Eizaguirre), marc.sanchez@urv.cat (M. Sánchez-Artigas).

https://doi.org/10.1016/j.jpdc.2023.104763

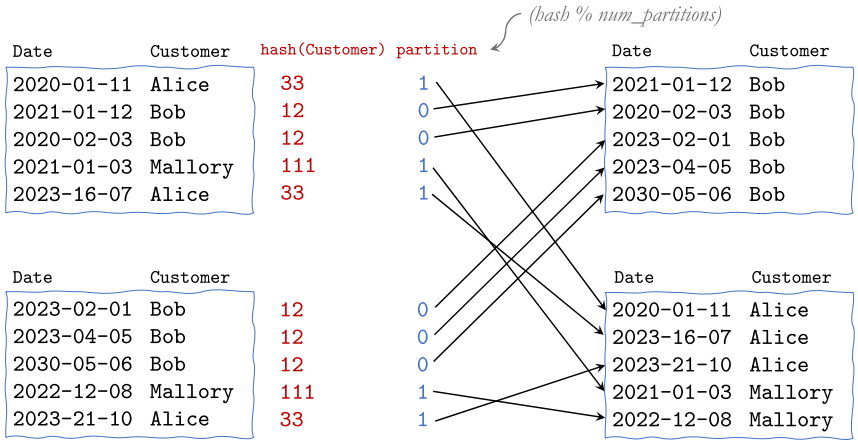


Fig. 1. Example of a data shuffle triggered by a `groupBy` operation on the column `Customer`.

not be directly implemented with serverless functions. Sharing intermediate data between serverless functions have become one of the major challenges in serverless data analytics [14,29,23,36,22].

As of today, the state-of-the-art practice for exchanging (intermediate) state between serverless functions is through remote storage. In practice, there are two types of remote storage systems that have been examined in the literature: 1. Elastic **far-memory** systems such as Pocket [21], Jiffy [19], and Crucial [6], which achieve low-latency (sub-millisecond), and 2. **Object storage** systems such as Amazon S3, which are much slower, but cheaper and optimized to deal with big data objects. Although far-memory storage systems are faster than disk-based object stores [29,6], and are a solid alternative to share intermediate data, they also have their disadvantages. For instance, they typically scale up the memory capacity by adding more virtual machines (VMs) at the data plane, which is a time costly operation and can take minutes depending on the instance type [21]. Moreover, VM scaling tends to act at a coarse granularity, which may lead to either system degradation or resource underutilization in the absence of prior knowledge [19,21].

Our approach. In this research work, we focus on object storage for one powerful reason: the need to develop a high-performance, "pure" serverless solution to serverless data shuffles. Object storage is available in all public clouds, and because it is an "always on" service, it incurs no start-up delay, thereby keeping the serverless principle of "no server management from the end users" intact. In principle, this makes it possible to design a pure serverless solution, but there are some knobs to tune to get the best out of object storage.

We propose in this work a smart serverless shuffle auto-tuner called Seer. More precisely, Seer chooses "on the fly" between two shuffle implementations to deliver improved efficiency: the direct solution, akin to the hash-based solution in Spark [44], and the multi-round shuffle introduced in [23], which trades off I/O throughput for latency. To go "serverless", Seer makes this decision based upon the evaluation of two performance analytic models: one for direct shuffle, and the other for the simplest incarnation of multi-round shuffling, i.e., the two-level, mesh-like algorithm presented in [23]. Importantly, the selection of the optimal implementation is made at runtime, and the only input these models need from a job is the volume of data to be shuffled at the current stage, which can be dynamically inferred as the sum of the individual data partitions. Simply put, Seer does not require jobs to know (even an estimate of) intermediate data sizes a priori. When a data shuffle operation is encountered, Seer dynamically determines the right shuffle implementation, along with the optimal degree of parallelism that minimizes shuffle time. The performance models require the characterization of the remote object storage service via a series of basic measurements. For this work, we characterized the performance of IBM Cloud Object Storage (COS), but we sense that other object stores behave similarly (e.g., S3 [29]).

Moreover, we integrate Seer into PyWren-IBM [31], a serverless data analytics engine that executes on IBM Cloud, and compare our integration to several state-of-the-art data analytics systems of distinct flavors, from serverful systems such as Spark to serverless cloud services such as Google BigQuery [9] and Google serverless Spark [13], and research prototypes such as Caerus [46] and SONIC [22]. Our results confirm the accuracy of our auto-tuning approach, but more importantly, attest that object storage is a practical but powerful approach for shuffling data. To give some first numbers, the performance of Seer is 5.6X better than serverless Spark in TPC-DS. Similarly, Seer is 6X faster than serverful Spark for the 100 GB Terasort benchmark for the same number of vCPUs. In comparison to other serverless systems, Seer also delivers better performance across all the workloads. Just to illustrate, we would have achieved up to 9.9X higher performance/$ (here performance denotes the inverse of latency) over SONIC+S3, being direct shuffle the optimal choice in this case. The comparison with SONIC is very alluring to practitioners because it examines the cost-efficiency aspect of Seer against a strong baseline. It must be noticed that for this workload, we also ran Seer over AWS S3 object store to gain a broader view of the problem. Interestingly, we found that the underpinning cost model of Seer for both IBM COS and AWS S3 is almost identical, which is useful to measure how Seer is able to improve data shuffles in different cloud providers.

Very interestingly, Seer is a slightly faster (13%) than Caerus, albeit Caerus uses Jiffy [19], a far-memory system. At least for us, this result calls in question the often quoted mantra that object storage is ill-suited for serverless shuffles [29,23], and advocates for more research.

Contributions. The key contributions of this paper are:

- Development of performance models for the direct and two-level shuffle methods (§4), and the performance characterization of the IBM COS service (§2). Using these models, we analyze their trade-offs, and show that no single method holds an edge over the other under all conditions, which motivates the need for a hybrid and dynamic approach.
- Design of Seer, a shuffle manager which dynamically chooses the optimal shuffle implementation for a given stage in a DAG without prior knowledge (§4).

- Integration of Seer with PyWren-IBM [31],[1] along with its evaluation against Spark and serverless analytics systems, including BigQuery and serverless Spark, and some research prototypes (e.g., Caerus and SONIC). In addition to latency, the comparison against some of these systems includes cost to assess the cost-efficiency of Seer. Our results demonstrate that Seer is cheaper, or similar in cost, than "serverful" alternatives such as Apache Spark and serverless systems such as BigQuery. (§5, §6).

A preliminary version of this work has appeared at ACM/IFIP Middleware'22 [34]. Compared to the conference version, we have extended the characterization of the IBM COS service, added a comparative analysis between both shuffle methods, and an evaluation against SONIC [22], a manager that selects the optimal data-passing method for each edge of a serverless DAG.

2. Preliminaries

2.1. Problem statement and system model

The core idea behind serverless data analytics is to use functions to dynamically adjust resources over time, and avoid both resource under- or over-provisioning, which lead to resource wastage or performance degradation in server-centric deployments, respectively. In this sense, serverless platforms facilitate fine-grained scaling of resources to match application needs.

Given the fine-grained elasticity of cloud functions, and serverless storage systems such as Amazon S3, or IBM Cloud Storage (COS), the main challenge in executing shuffles is:

> **Problem statement:** *Given the fine-grained scaling of compute resources, can we find a way to determine at runtime the optimal number of functions (degree of parallelism) that better utilizes remote object storage without extra knowledge at job submission time?*

This is one of the main goals pursued by Seer: being the "living proof" that it is possible to tune shuffles at runtime in order to maximize the I/O efficiency of object storage.

Since Seer optimizes shuffle operations, it is important to define what we understand by "parallelism" in the context of serverless data shuffling. Compute parallelism will be of course leveraged by function invocations, which will we refer to in this paper as "workers". Nonetheless, we will limit the inner parallelism of workers to 1 vCPU, which can be achieved in IBM Cloud Functions by provisioning 2,048 MB of RAM to the workers. The reason for this is two fold. On the one hand, the equivalent of a single core "worker" can be found in all FaaS platforms under different memory allocations, which opens the door to a potential generalization of our insights to other cloud providers and systems. On the other hand, a single core prevents the use of well-known optimizations to achieve improved I/O efficiency, such as the exploitation of intra-worker parallelism to overlap computation and communication, and so far hide the access latency to remote storage. In this sense, our results are conservative, and should be interpreted as a baseline with room of improvement. Despite the use of single-vCPU workers, Seer is able to outperform other systems with more CPU cycles and memory allocation (see §6).

As a result of the provisioning of all workers to access 1 vCPU, we will be able to unambiguously refer to the number of workers that concurrently run (in a shuffle operation) as the degree of parallelism. As typical in data analytics frameworks (e.g., Spark), we will assume that operations with wide dependencies [44] like `groupBy` will trigger a shuffle that Seer will optimize.

2.2. Object storage

To be able to choose the most appropriate configuration of shuffle algorithm and parallelism, it is crucial to characterize the object storage performance to ascertain if it is possible to build useful analytical models for the shuffle algorithms. Unsurprisingly, the answer is "yes" and can be achieved in terms of I/O **throughput** and **bandwidth** as usual. Although our focus is on IBM Cloud Storage (COS), this characterization is general enough to encompass other object stores such as Amazon S3, Google Cloud Storage, and Azure Blob Storage. All the measurements of this section were conducted on the `us-east` region.

(I/O Throughput). In this context, read or write throughput refers to the rate of read or write operations to an object store. Concretely, we measure the throughput as the number of read or write operations per second, abbreviated as OPS/s.

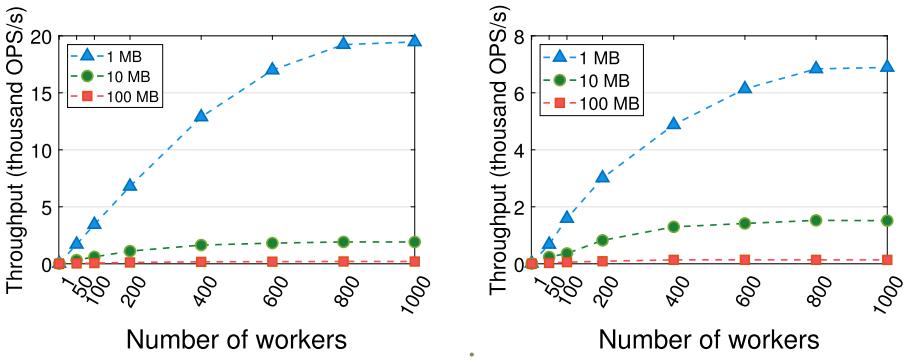
Object stores have not been conceived to deliver high throughput for small files, and indeed public cloud vendors impose request limits on them [4,12,5]. To wit, as of April 2022, the rate limit on Amazon S3 is of 3.5k write and 5.5k read requests/sec per prefix in a bucket. The first interesting observation for IBM COS is that there is no limit on the number of requests/sec. Consequently, we had to measure how much **throughput** functions can glean from IBM COS. As a first result, we did not observe any significant improvement by increasing the number of buckets and prefixes. Specifically, we found that the only two factors dominating throughput were the object size (s) and the degree of parallelism (p). To give a sense of how these two factors affect the I/O throughput, we benchmarked the read and write throughput as we changed the degree of parallelism for several object sizes.

Throughput results are shown in Fig. 2 for the maximum possible memory size. As expected, throughput decreases as the object size inflates, with the read throughput being much higher than the write throughput, thereby showcasing strong asymmetry. In all cases, I/O throughput scales linearly up to a certain parallelism, where the curve begins to flatten out. The flattening point depends exclusively on the request size. This behavior makes it easy to model the throughput curve as function q of two variables $q(s, p)$, where for low degrees of parallelism the curve exhibits a linear behavior. Likewise, the non-linear part is smooth, so it can be well approximated with a small number of measurements. In Fig. 4, we illustrate the fitted throughput curves for 10 MB-sized files as an example. Both curves were built from only 27 measurement samples (3 probes per parallelism degree) with a total measurement cost of $3.3 each curve (each worker ran for 15 seconds).

(Bandwidth). Fig. 3 depicts the per-worker bandwidth (in MB/s) as we vary the degree of parallelism and the request size. From this figure, we find a very important result: as the degree of parallelism increases, the read bandwidth almost does not change for up to 400 workers, and only the per-worker write bandwidth degrades slightly for large request sizes. This result is good news, since it allows us to treat the per-worker bandwidth as a constant in the analytical models developed in §4.3 for up to 400 workers with high accuracy, and therefore, simplify our equations.

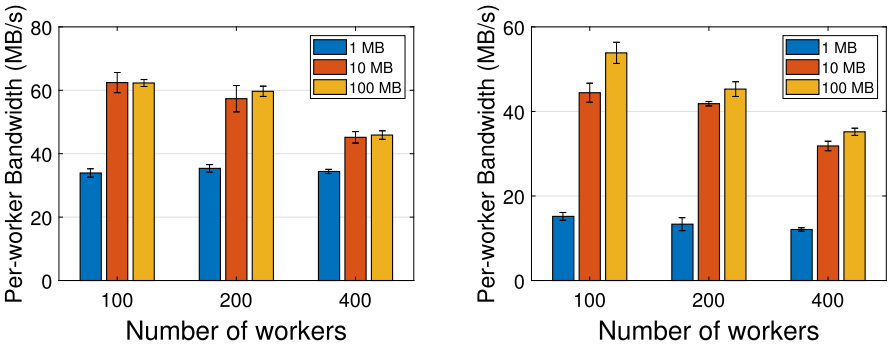
(Effect of the worker memory size). To glean more insight into the performance of object storage, we studied the effect of worker memory size on throughput. Specifically, we quantified its influence from two distinct angles. Firstly, we did so by varying the degree of parallelism. Then, we measured its impact by modifying the file size. Fig. 5 reports the results for 10 MB-sized file requests when the degree of parallelism is increased. Non-surprisingly, as

[1] now recast as Lithops [32,33].



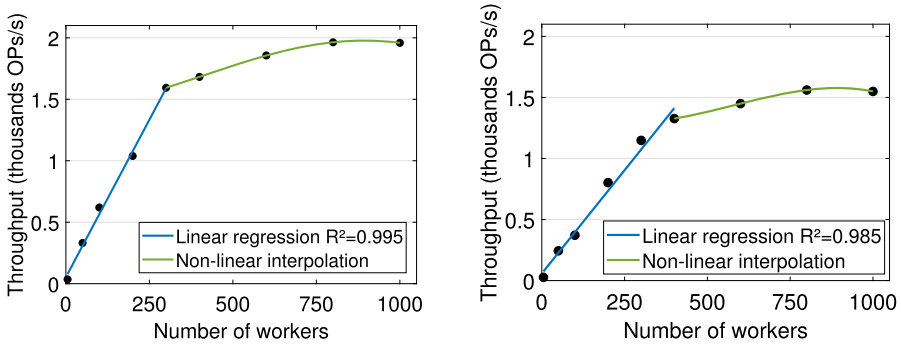
(a) **Read throughput**. (b) **Write throughput**.

Fig. 2. Measured read/write throughput (thousand OPS/s) for increasing parallelism and file sizes.



(a) **Per-Worker Read BW**. (b) **Per-Worker Write BW**.

Fig. 3. Measured per-worker read/write bandwidth (in MB/s) for increasing parallelism and file sizes.



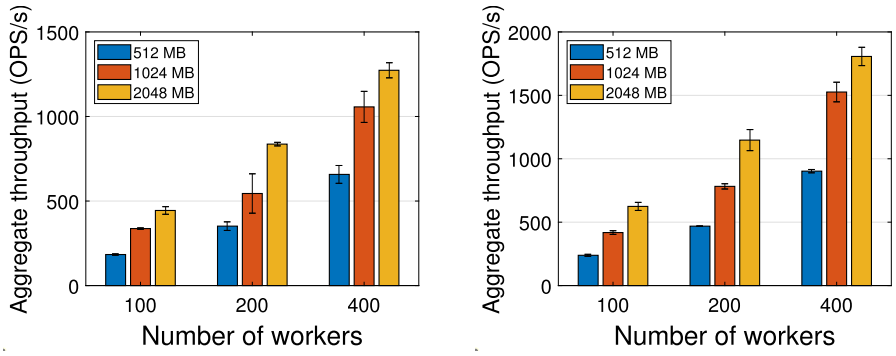
(a) **Read throughput**. (b) **Write throughput**.

Fig. 4. Fitted throughput curves $q(10\text{ MB}, p)$.

CPU power is allocated proportionally to the amount of RAM provisioned, both read and write throughput is maximized for 2 GB-sized serverless workers. However, the increment in throughput is sub-linear. The most striking observation, though, is the fact a large number of small workers is not always ideal. As can be seen in Fig. 5, it is better off having 100 workers of 2 GB of memory than 200 workers, each sized with 0.5 GB of RAM.

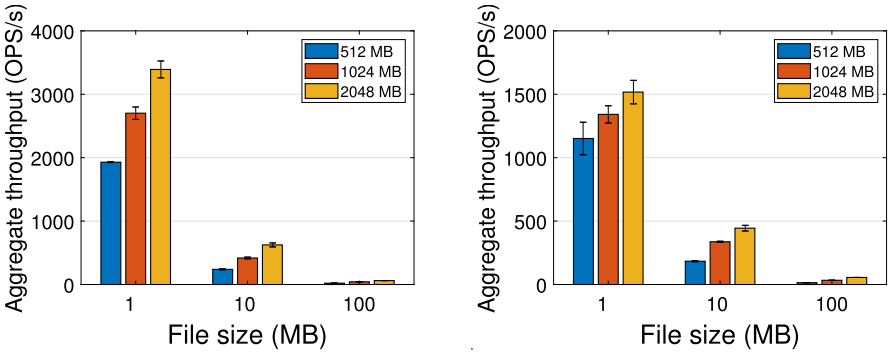
Nevertheless, the sub-linear advantage of a larger memory allocation may make more cost-optimal the exploitation of a larger fleet of smaller memory workers. We say "may", because as parallelism increases, the size of the intermediate files being generated during the shuffling process decreases, which could worsen performance. To better understand this, let us go back at Fig. 2 and multiply the object size with the throughput to approximate the aggregated bandwidth. Then, it is easy to see that using smaller object sizes (e.g., of 1 MB) means a lower aggregated bandwidth and worse performance. This finding is well-aligned with previous work [29]. Further, sub-linearity of memory size is also maintained across object sizes as can be seen in Fig. 6.

(Practical takeaway). As a practical result of the above analysis, characterizing the performance of object storage to store shuffle data is practical. However, we observed that worker memory size affects throughput, which is maximized for the maximum memory allocation of 2 GB for IBM Cloud Functions. This result is not surprising, since CPU power is allocated proportionally to the amount of RAM provisioned. What might complicate matters is the treatment of memory allocation as an optimization variable. Although the above characterization proves useful for any worker memory size, memory allocation and I/O throughput exhibit a sublinear behavior, which was also observed in [29] for AWS Lambda. For instance, we got a read throughput of 782 requests/sec for 1 GB-sized workers in a shuffle operation with a degree of parallelism of



(a) **Read throughput**. (b) **Write throughput**.

Fig. 5. Measured read and write throughput (in OPS/s) for different memory allocations as parallelism is varied.



(a) **Read throughput**. (b) **Write throughput**.

Fig. 6. Measured read and write throughput (in OPS/s) for different memory allocations as object size is varied.

200 workers. Doubling the memory size, only improved this quantity by 1.47×.

3. Shuffling methods

3.1. Direct shuffle

The direct shuffle algorithm is the most preponderant in the serverless literature, with many research works taking it up as the default all-to-all data passing method for intermediate data [29,23,36]. This method has a simple description: each of the p workers holds a split of the input data in memory, and uses a partitioning routine to divide it into p partitions, one per receiver (e.g., based on the hash value of the records from the input split). Whatever the partitioning routine, each worker writes the data of each partition into an independent file following a naming convention that typically reflects its own ID along with the ID of the receiving worker of the file. Finally, each worker reads all files that include its own ID as a receiver. Since the "source" workers may be slower than a receiver, a receiver must watch for a file until it exists.

The problem with this method is that the total number of files is quadratic in p: each of the p workers reads from and writes to p intermediate files. Simply put, as the parallelism degree raises, the quadratic number of requests may saturate or cause throttling of the object store throughput. For example, consider the shuffling of 10 TB data. As noted before, cloud functions have stringent resource limitations that restrict the amount of data a serverless worker can process (e.g., up to 2 GB of RAM for IBM Cloud Functions). Assuming 1 GB input splits to maximize memory utilization, this will result in 10^4 partitions, and 10^8, or 100 *million*, intermediate files. Considering a write throughput of 10 thousand requests/sec, just writing all the intermediate files would take around 2.7 hours, which is impractical in terms of runtime complexity. However, our validation results show that this method could be a compelling solution to the serverless shuffling problem for many production workloads (e.g., found at Microsoft or Google [26]). Loosely speaking, although the input data may be big in production, the amount of data shuffled is between 5-10X less than the amount of input data [26]. As seen in [45], very often the limiting factor encountered in many production workloads is that number of map tasks, say M, and reduce tasks, say R, is oversized in relation to the shuffle data size D, yielding intermediate files of a few kBs: $\left(\frac{D}{M\times R}\right)$.

With Seer, however, we demonstrate that direct shuffling over remote object storage can outperform VM deployments with Spark for medium scales (e.g., a few GBs), provided that the optimal number of workers is used. Recall that for direct shuffling the total number of I/O requests is $\Theta\left(p^2\right)$. We have seen that it is possible for many practical shuffle data sizes to find a p value that leads to sufficiently large intermediate files $\left(\frac{D}{p^2}\right)$ that can: 1. Effectively exploit the high bandwidth of object storage, and 2. Maintain the quadratic pressure on throughput at bay. Contrary to prevailing wisdom, we have found that this simple form of shuffling is not a bottleneck for many jobs, in a similar way as the network has ended up being not so important in "serverful" data analytics [26].

3.2. Two-level shuffle

As just discussed above, for larger shuffles it is vital to reduce the request complexity, so that the throughput requirements do not lead to degraded performance. We observe that even if a single file with all intermediate partitions is produced per "source"

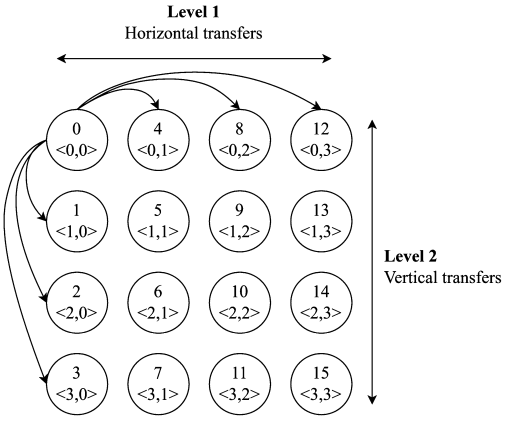


Fig. 7. Two-level shuffle communication pattern.

Algorithm 1 Two-level shuffle algorithm.

Input: x: **int** (worker ID); p: **int** (degree of parallelism);
$\mathcal{D}_x$: **InputSplit** (x's input data partition)
Require: $p \in \mathbb{N}_0$ s.t. $(\sqrt{p})^2 = p$; $0 \leq x < p$.
$\langle x_h, x_v \rangle \leftarrow F(x)$ ▷ (x's projection to the logical grid)
$\mathcal{H}_x \leftarrow \{y | 0 \leq y < p : y_h = x_h\}$ ▷ (x's horizontal group)
$\mathcal{V}_x \leftarrow \{y | 0 \leq y < p : y_v = x_v\}$ ▷ (x's vertical group)
tmp $\leftarrow$ DirectGroupShuffle($x, \mathcal{H}_x, \mathcal{D}_x$)
result $\leftarrow$ DirectGroupShuffle($x, \mathcal{V}_x$, tmp)

worker (e.g., as it happens in Spark [44][2]), the total number of concurrent read or fetch requests still amounts to p^2, bottlenecking the system for large p values [45].

To circumvent this bottleneck, [23] presented a multi-level shuffle operator that needs $\mathcal{O}(\sqrt[k]{p})$ requests to exchange all the data among the p workers, where k refers to the number of levels. This approach, well-known in the HPC literature, consists of *logically* structuring the serverless workers into a k-dimensional mesh with side length $\sqrt[k]{p}$, and use an all-to-all collective operator k times, once for each dimension to realize the full exchange of data among the workers.

The fundamental difference with a parallel architecture is that the all-to-all operator cannot use direct communication between the group of workers in each dimension. For this reason, the all-to-all operator is simply replaced in [23] by the direct shuffle method in all the k dimensions. As a result, the number of requests will increase sub-quadratically with the dimension size (essentially, $p\sqrt[k]{p}$), rather than quadratically with the number of workers.

In this work, we will employ only the two level version, as it suffices for our purposes here to examine "when" the multi-level approach is actually effective. Then, this algorithm can be simply implemented by projecting the set of workers onto a logical two-dimensional mesh, and let each worker firstly perform a horizontal exchange followed by a final vertical shuffle. For a $\sqrt{p} \times \sqrt{p}$ lattice, the projection function is as simple as $F(x) := \langle x_h, x_v \rangle := \langle x \mod \sqrt{p}, x/\sqrt{p} \rangle$. Simply put, $F(x)$ projects a worker's ID $x, 0 \leq x < p$, onto coordinates $\langle x_h, x_v \rangle$, where "mod" and / are the modulo and integer division, respectively. We refer to DirectGroupShuffle($\cdot$) as the invocation of the direct shuffle method for the group of workers in each of the two dimensions with respect to a worker's ID (see Fig. 7). Algorithm 1 describes the two-level approach under this notation.

Although efficient to reduce throughput requirements, the two-level shuffle method is not always the best option due to the two full scans of the input data (see §4.3 for details). Our model, however, allows us to anticipate "when" this method will be the most efficient to minimize the shuffle time, falling back to direct shuffling in the event of "augured" insufficient performance. Table 1 and Table 2 compare both methods.

[2] In Spark, at the end of any map-side shuffle task, a pair of files is produced, one for the shuffle data and other to index shuffle blocks in the former.

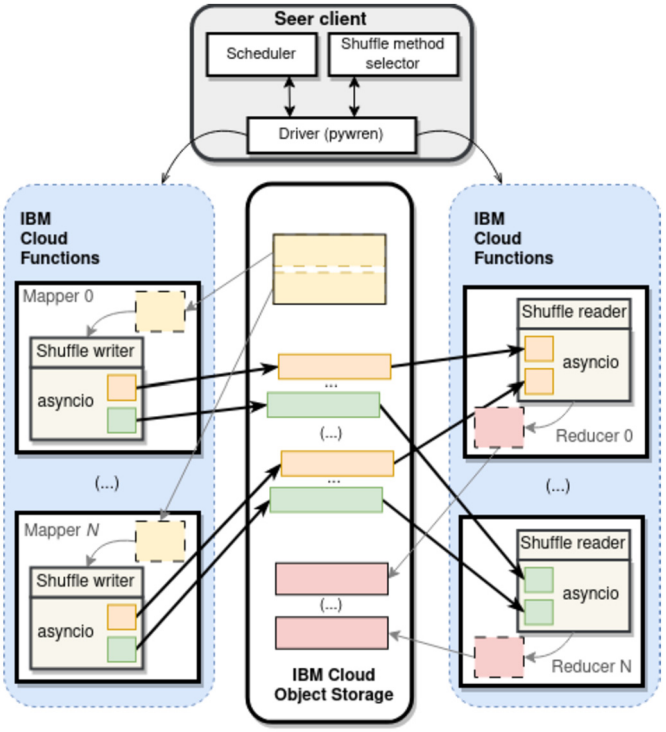


Fig. 8. Architecture overview of Seer. Example of a data shuffle.

Table 1
Comparison of the shuffle methods. D refers to the shuffle size, p the number of workers.

Method	# of reqs.	request size	# of scans	shuffle time
Direct	$2p^2$	$\Theta\left(\frac{D}{p^2}\right)$	1	Eq. (1)
Two-level	$4p\sqrt{p}$	$\Theta\left(\frac{D}{p\sqrt{p}}\right)$	2	Eq. (2)

4. Design

4.1. Architecture overview

For Seer, one of our main design goals is to use only existing serverless components. Fig. 8 depicts its high-level architecture. The Seer client runs on the local development machine of the data scientist. It has been built upon PyWren-IBM [31]. But it reworks part of its base code to introduce additional modules, such as a stage scheduler and our smart shuffle method selector. More precisely, PyWren has been leveraged to invoke a (potentially large) number of serverless workers, who execute the analytics job in a data-parallel manner. Fig. 8 shows an example of a data shuffle operation. During a data shuffle, workers communicate exclusively through remote object storage (e.g., IBM COS). We recall that a data shuffle operation involves a pair of stages: the current stage where each worker (or "`mapper`") performs a shuffle write at the end, and the subsequent stage, where each worker (or "`reducer`") reads the needed intermediate partitions through a shuffle read operation. For both shuffle writes and reads, Seer borrows the `asyncio` library [7] to achieve high concurrency for I/O-heavy tasks.

The shuffle selector module is responsible for choosing the optimal shuffle implementation and level of parallelism per stage. This selection is made at runtime from only the total volume of data

Table 2
Comparative analysis of both shuffle algorithms for different configurations of workers and data sizes. The check marks identify the dominant factors: throughput (abbreviated "Tput.") and bandwidth (abbreviated "BW") that limit the performance of each algorithm according to equations (1) and (2). The shadowed cells show the configurations where direct exchange is better (Speedup < 1).

Data size (GB)	# of workers	Dominant Factors								Speedup $\left(\frac{T_{direct}}{T_{2level}}\right)$	
		Direct				2-level					
		write Tput.	write BW	read Tput.	read BW	write Tput.	write BW	read Tput.	read BW	Predicted	Empirical
1	9	✓			✓		✓		✓	0.47	0.53
16	49		✓		✓		✓		✓	0.63	0.65
32	196	✓		✓		✓		✓		1.32	1.24
40	289	✓		✓			✓		✓	1.84	1.74

to be shuffled. When a data shuffle operation is found, the Seer client calls the selector module to dynamically determine the right shuffle implementation (see §4.5 for further details).

Job execution is managed by a separate scheduler module. Based on an intuitive annotation API, the scheduler extracts the dependencies between stages, parses data types and disseminates execution parameters. Stage execution is performed in topological order based on the dependency tree. That is, independent stages are executed in parallel, while dependent stages are run serially but in a pipelined fashion (see §5 for details).

4.2. Job execution

After the introduction of the main components of the architecture, we are now in position to portray a typical execution of an analytics job with Seer. It involves the following five steps: ❶ The user starts up the Seer client. Once up and running, the scheduler module in the Seer client elicits the dependencies between stages and parses the data types from the DAG representation of the job; ❷ Then, the scheduler starts to execute the stages in topological order based upon the dependency tree. As in other analytics frameworks such as Spark [44], Seer assumes that each stage concludes with an operator that requires a data shuffle (e.g., a `sort`). The scheduler uses these breakpoints to determine the appropriate level of parallelism for optimally shuffling the intermediate partitions. ❸ Concretely, this is achieved by calling the shuffle selector module when the breakpoint at the end of each stage is found. ❹ With the optimal level of parallelism, the scheduler leverages the underlying PyWren-IBM API to invoke the corresponding number of serverless workers. ❺ Finally, the exchange of the intermediate data takes place according to the chosen shuffle algorithm by the shuffle selector module.

4.3. Performance models

As argued in the literature, tuning shuffle in frameworks such as Spark is hard to achieve without tedious experimentation [45, 39] for several reasons. To begin with, Spark materializes the intermediate shuffle data on disks for better fault tolerance, and it is thus exposed to HDD efficiency issues, e.g., due to small random disk reads. This makes the tuning of shuffle in Spark to have to consider many factors, such as the existing trade off between shuffle and disk spilling I/O efficiency [45]. Fortunately, object storage performance is much easier to characterize as seen in §2.2. For a fixed worker memory size, there exist representative bandwidth limits on the one hand, while I/O throughput can be approximated pretty well with a few measurements on the other. We want to point out here that our goal with Seer is not to estimate exact shuffle times, but to provide an *online* method to choose the most adequate shuffle algorithm along with its optimal number of workers. Our evaluation results verify that Seer attains by far this goal.

For our models, we assume that the volume of data to be shuffled through serverless storage is known and amounts to D bytes. Details of how D can be estimated are given in §5.

(Performance factors). As demonstrated in §2.2, there are two factors that limit the performance of object storage systems: the **per-worker bandwidth** and **throughput**. As observed in §2.2 and prior measurements [29], per-worker bandwidth is typically asymmetric, for we denote the maximum available per-worker read and write bandwidth as b_r and b_w, resp. We also assume that there is a limit on the number of read and write operations/sec. Recall that these limits are determined by multiple factors, including the 1. **Blob size**, and 2. **Degree of parallelism**, for a given worker memory size. This leads to throughput limits that are non-linear functions of these factors, for we represent these limits as $q_r(s, p)$ and $q_w(s, p)$, respectively, where s refers to the blob size in bytes.

For simplicity, we assume that the per-worker read and write bandwidth limits, namely b_r and b_w, respectively, are valid as the number of workers or parallelism increases. Although this is not always the case, we empirically found that I/O throughput is the dominating bottleneck for large parallelism. Put differently, we assume that the available per-worker bandwidth stays constant for the parallelism level where throughput is not the major bottleneck. This is clearly seen in Fig. 3 that shows little or no degradation in the bandwidth for up to 400 workers.

(Direct shuffle). In the direct approach, each worker first writes its input split of size $\frac{D}{p}$ bytes to object storage by issuing p concurrent requests of average size $\frac{D}{p^2}$. Given that we must perform p^2 write requests with an aggregated throughput of $q_w\left(\frac{D}{p^2}, p\right)$, the time to complete these requests can thus be approximated by $\frac{p^2}{q_w(D/p^2, p)}$ when throughput is the major system bottleneck. Analogously, assuming that the per-worker write bandwidth limit b_w is the main bottleneck, the time to complete these requests becomes $\frac{D}{b_w \times p}$. Finally, considering both potential bottlenecks, the time it takes the write stage of direct shuffling is:

$$T_w = \max\left\{\frac{D}{b_w \times p}, \frac{p^2}{q_w\left(D/p^2, p\right)}\right\}.$$

By evident dual reasoning, it can be easily seen that the read stage takes $T_r = \max\left\{\frac{D}{b_r \times p}, \frac{p^2}{q_r(D/p^2, p)}\right\}$. Thus, the shuffle time of the direct method is given by:

$$T_{direct} = T_w + T_r = \max\left\{\frac{D}{b_w \times p}, \frac{p^2}{q_w\left(\frac{D}{p^2}, p\right)}\right\} + \max\left\{\frac{D}{b_r \times p}, \frac{p^2}{q_r\left(\frac{D}{p^2}, p\right)}\right\}. \quad (1)$$

Typically, for a small degree of parallelism, direct shuffle is limited by the per-worker bandwidth. But as the amount of data to shuffle grows, so as the number of workers needed to store the input data due to the memory limits of functions (e.g., up to 2 GB in IBM

Cloud Functions), throughput ends up dominating the shuffle time method as stated in Eq. (1).

(Two-level shuffle). In essence, the two-level approach can be interpreted as an execution of direct shuffle two times in a row, where each worker first communicates with the subset of workers in its horizontal group, and then, within the members of its vertical group. The practical interest of this approach is that the total number of requests in each phase increases only quadratically with the horizontal and vertical group sizes and not with the number of workers. For optimality, the two-level shuffle method assumes symmetric group sizes, that is, both horizontal and vertical groups have a size of $\sqrt{p}$ workers. So, to derive the optimal shuffle time for this method, one mostly needs to properly plug in the factor $\sqrt{p}$ in Eq. (1). Observe that this method consists of four stages: a write stage followed by a read stage to accomplish horizontal group communication, plus another write-read stage for the vertical data exchanges. Altogether, this leads to a total shuffle time of:

$$T_{2level} = T_{horizontal} + T_{vertical} = (T_w + T_r) + (T_w + T_r)$$
$$= 2Tw + 2T_r = 2 \times \max\left\{\frac{D}{b_w \times p}, \frac{p\sqrt{p}}{q_w\left(\frac{D}{p\sqrt{p}}, p\right)}\right\} +$$
$$2 \times \max\left\{\frac{D}{b_r \times p}, \frac{p\sqrt{p}}{q_r\left(\frac{D}{p\sqrt{p}}, p\right)}\right\}. \quad (2)$$

Two important subtleties arise from Eq. (2) and this method. Firstly, *it reads and writes the input data two times*, instead of just one. That is, a full transfer of the input data takes place in each of the two levels, which may increase the running time, though each level performs better than direct shuffle thanks to the reduction in the number of I/O requests. Second, *such a reduction comes at the expense of an increased request size*. More concretely, this method performs exactly $p\sqrt{p}$ requests in each of its four stages, which means a fall-off factor of $\sqrt{p}$ compared to direct shuffling. This reduction factor is, however, transfered to the size of requests, which increases by a factor of $\sqrt{p}$. While the average request size for direct shuffling is of $s_{direct} = \frac{D}{p^2}$ bytes, it grows to $s_{2level} = \frac{D}{p\sqrt{p}}$ for the two-level method. Simply put, the two-level algorithm decreases the number of requests but makes them heavier. Since I/O throughput degrades as the object size increases, the time of each of the two levels may not necessarily be a 50% shorter than the overall time taken by direct shuffling. Jointly, these subtleties blur the line between direct shuffle and two-level shuffle, and as we shall see in §6.

4.4. Comparative analysis

In Table 1, we report a summary of the differences between both shuffle algorithms. Succinctly, the major issue of direct shuffle is the steep growth in the number of requests, which is quadratic on the size of the worker pool. Nevertheless, it only requires a full scan of the dataset compared to the two-level algorithm, which reads and writes the input data two times. Recall that the two-level algorithm needs to horizontally exchange a full copy of the input dataset. And once this is materialized, it proceeds with a vertical exchange of the partially redistributed data across the $\sqrt{p}$ horizontal groups to create the final partitions. We see that the $\sqrt{p}$ fall-off factor in the total number of requests will only be useful when the two scans on the input data do not become the dominant factor. To put it bluntly, while each of the two exchange phases in the two-level approach, namely horizontal and vertical, can be individually faster than the single-phase direct shuffle, their sum can take longer, blurring the boundaries between both algorithms.

To shed light on this matter, we have used our performance models to determine the most effective shuffle algorithm for different configurations, and pinpoint what factor is limiting the performance of each shuffle algorithm. Furthermore, we have leveraged each configuration and executed an equivalent real experiment to validate our analytical models. Concretely, we have issued a `groupBy` query on four metabolomics datasets from METASPACE [27], and recorded the empirical speedup of the two-level method compared to direct shuffle: Speedup $:= \left(\frac{T_{direct}}{T_{2level}}\right)$. Accordingly, speedup values below unity mean that the direct approach is better, while speedup values greater than one mean that the two-level approach is more efficient. Performance results are given in Table 2.

The first interesting insight is that our performance models estimate general trends pretty well. For all the configurations, the predicted speedup values are almost identical to the empirical ones with a margin of error below 15% in all cases. This opens the door to the exciting avenue of leveraging our performance models to automatically identify the most performant combo of shuffle algorithm and parallelism (see §4.5). The difference in time between the estimated and real values can be attributed to the fact that we do not account for certain overheads such as the variance of IBM COS, which is a globally shared service; serialization latency; data formats; data skewness, etc. In spite of this, we believe that our models are sufficiently accurate to make informed scheduling decisions.

The other key observations are the following:

- For a few tens of workers, or equivalently stated, when I/O throughput is not the limiting factor, direct exchange is faster than the two-level approach. For the dataset of 16 GB and 49 workers, direct shuffling spent 33.24 ± 0.71 seconds versus the 49.01 ± 0.64 spent by the two-level method. This is mostly due to the fact that two-level shuffling transfers the dataset two times. Indeed, the dominant factors in the two-level method confirm this fact. As listed in Table 2, the read and write bandwidth were the two factors that bottlenecked the two-level method in this case. Interestingly, each individual phase in two-level shuffling took less than direct shuffle, i.e., around 21 seconds per phase. But altogether they took much more time.
- Related to the above observation, if we cross-compare the performance of both algorithms for the 16 GB dataset against the smallest one of 1 GB, another interesting insight ensues. Specifically, the returns of the two-level approach over the direct shuffling are sublinear both in the number of workers and the dataset size. Despite increasing the data size from 1 GB to 16 GB, the ΔSpeedup is only of 0.12. The reason for this result is that the required aggregate bandwidth per phase is the same as in the direct method, while it is the request size that increases. In the direct method, the average request size is of $\mathcal{O}\left(\frac{D}{p^2}\right)$ bytes, growing to $\mathcal{O}\left(\frac{D}{p\sqrt{p}}\right)$ bytes for the two-level approach. Hence, the penalty introduced by a higher number of write and read requests in the direct algorithm (i.e., $49^2 = 2,409$ write and read requests) is to a certain extent compensated by the two scans of the two-level approach.
- Only when the scale grows to hundreds of workers, the two-level method increasingly becomes better, and clearly holds an edge over the direct approach for 289 workers. It is worth to note here that for 289 workers the limiting factor of direct shuffle is throughput, whereas the limiting factor of the two-level approach becomes the available bandwidth between

the object store and the workers, as pointed out by the check marks in Table 2. Since I/O bandwith is not infinite, sooner or later, the per-worker bandwidth will not be longer constant, thereby preventing the two-level shuffle from scaling as expected due to the two full scans of the input data. As shown in Fig. 3, the per-worker bandwidth starts to level-off and deviate from the linear trend around 400 workers in IBM COS. Indeed, the maximum speedup we recorded was of 4.1X in our experiments.

(Practical takeaway). To wrap up, there is no one-size-fits-all solution to shuffle data through remote object storage, for which prediction tools, e.g., based on our performance models, are "de rigueur" to optimize latency. This is vital in workflows comprised of multiple stages, where each stage may have a specific amount of data to exchange, so that the appropriate parallelism along with the correct shuffle strategy can make a big difference.

4.5. Shuffle method selection

One important benefit of our models is that they can be employed, for instance, as a basis to select the optimal shuffle algorithm for each stage of a DAG execution plan. Remember that in standard data analytics tools (e.g., Spark), the intermediate data between stages is transferred via the shuffle operation, so the appropriate choice of the shuffle method can trigger significant savings. The "beauty" of our models is that they only require the size of the data to shuffle at the end of each stage as per-job information. Thus, Seer permits to employ *late binding* to delay the decision of which shuffle algorithm to apply until a shuffle is scheduled at the end of the stage. These are good news for several reasons: 1. **Practicality**, since there is no need to profile the DAG to determine the shuffle data size for every stage; only the storage parameters of the remote storage service are necessary; and 2. **Optimality**, as it makes possible to efficiently reach a global optimum making local selections at each stage without prior user information at job submission time.

Note that for a specific shuffle data size D, the search space of parameter p in both Eq. (1) and Eq. (2) is discrete and bounded. More specifically, the search space reduces to the interval $p := \left\lceil \frac{D}{w} \right\rceil, \left\lceil \frac{D}{w} \right\rceil + 1, \ldots, \mathcal{P}$, where w denotes the worker memory size, and $\mathcal{P}$ is the maximum concurrency limit (e.g., of 1k activations for IBM Cloud Functions). This ensures that the global minimum can be found after just a few evaluations of Eq. (1) and Eq. (2). The search space is so constrained (at most a few thousand evaluations) that locating the global minimum, even when using an inefficient linear search mechanism, has negligible impact on system performance in comparison to the much longer shuffle time.

Algorithm 2 reports the pseudocode of our selection method for minimizing the shuffle time in a single stage. The algorithm is self-explanatory. First off, for each shuffle method j, the optimal degree of parallelism $\widehat{p}_j$ and shuffle time $\widehat{T}_j$ is found by searching over the discrete space $\mathcal{R} := \left\{ \left\lceil \frac{D_i}{w} \right\rceil, \left\lceil \frac{D_i}{w} \right\rceil + 1, \ldots, \mathcal{P} \right\}$. Next, the algorithm computes the speedup of the two-level method compared to direct shuffle: $\varphi = \frac{\widehat{T}_{direct}}{\widehat{T}_{2level}}$, where $\widehat{T}_{direct}$ and $\widehat{T}_{2level}$ are the estimated minimum shuffle times of the direct and two-level methods, resp. Finally, the two-level method is used for $\varphi > 1$, or Algorithm 2 falls back to the direct shuffling otherwise.

Interestingly, if for whatever reason, the parallelism is fixed for a given stage, or throughout the entire DAG, choosing the optimal shuffle algorithm takes only $\mathcal{O}(1)$ time per stage, instead of $\mathcal{O}(\mathcal{P})$. This occurs, for instance, in query processing engines such as Spark SQL [3], which employs a default number of 200 partitions for aggregations and joins. In this case, Eq. (1) and Eq. (2) are evaluated with the specific D and p values (e.g., $p = 200$ in Spark SQL), and the algorithm with the smallest shuffle time is chosen.

Algorithm 2 Shuffle method selection algorithm.

Input: D_i: **int** (shuffle data volume at stage i of the DAG); $\mathcal{P}$: **int** (maximum allowed concurrency); w: **int** (worker memory size); $\mathcal{S} := \langle b_r, b_w, q_r, q_w \rangle$: **tuple** (storage parameters)
Begin:
$\mathcal{R} \leftarrow \left\{ \left\lceil \frac{D_i}{w} \right\rceil, \left\lceil \frac{D_i}{w} \right\rceil + 1, \ldots, \mathcal{P} \right\}$ ▷ (p's search range)
$\langle \widehat{T}_{direct}, \widehat{p}_{direct} \rangle \leftarrow$ ExhaustiveSearch($\mathcal{R}$, Eq. (1), $\mathcal{S}$)
$\langle \widehat{T}_{2level}, \widehat{p}_{2level} \rangle \leftarrow$ ExhaustiveSearch($\mathcal{R}$, Eq. (2), $\mathcal{S}$)
Calculate $\varphi = \frac{\widehat{T}_{direct}}{\widehat{T}_{2level}}$ ▷ (Normalized speedup)
if $\varphi > 1$ **then**
  result $\leftarrow \langle$"Two-Level", $\widehat{p}_{2level} \rangle$
else
  result $\leftarrow \langle$"Direct", $\widehat{p}_{direct} \rangle$
end if

5. Implementation

The Seer prototype has been built on top of PyWren-IBM [31],[3] an extension of PyWren [16] to run Python-based analytics jobs on top of IBM Cloud Functions. While PyWren allows users to implement custom functions to execute shuffles with other cloud services, it lacks an actual shuffle operator, as noted by their authors in [29]. Likewise, PyWren-IBM [31] does not support shuffling, but provides a built-in `mapreduce` operator where the output from all the parallel `map` function instances is collected onto a single `reduce` function. So, Seer augments PyWren with support for data shuffles. The implementation includes the two shuffle algorithms, the models to select the optimal shuffle approach, and the set of operators (e.g., `sort` and `groupBy`) needed to implement the different benchmarks and analytics applications. The source code and artifacts are available at: https://github.com/GEizaguirre/seercloud.

(Job scheduling). PyWren is not equipped with a scheduler. In other words, it only supports the parallel execution of plain Python functions. Therefore, end users are given the responsibility to manually specify the job execution plan by writing code. Since the objective of Seer is not to design a fully-fledged scheduler (see, for instance, Caerus [46] for that purpose), we provide a simple annotation API that users can employ to specify the stage information Seer expects. The following is an example of a Seer job:

```
from seercloud.scheduler import Job
from seercloud.operation import Scan, Exchange, Sort

job = Job(num_stages = 2)
job.add(stage = 0, op = Scan , file = "Terasort_1GB.csv",
     Bucket = "seer-datasets")
job.add(stage = 0, op = Exchange)
job.add(stage = 1, op = Sort, key = "c0")
job.dependency( parent = 0, child = 1)
job.run()
```

(Shuffle execution plan). In terms of execution, we map the logical execution plan (i.e., the shuffle write and read tasks) of each shuffle method into a physical plan of two stages. For the two-level shuffle method, this means that the first three logical phases (i.e., the write and read tasks of the first level, and the write tasks of the second level, in this specific order) are mapped to the first physical stage, deferring the pending shuffle read phase to the second stage. This design facilitates their potential integration of tools like `WholeStageCodegen` [24] into Seer, which fuses operators as much as possible into a single task to maximize performance. The generated code in these tools is typically composed of blocks

[3] now recast as Lithops [32].

separated by pipeline breakers, where each block corresponds exactly to a stage in our annotation model.

For fast intermediate data (de)serialization, we use PyArrow,[4] a Python extension library for the C++ Arrow framework. We also use Snappy compression to reduce data size. Due to the absence of intra-level parallelism within workers [23] and the infamous global interpreter lock (GIL), we leverage the Python's `asyncio` module [7] to handle concurrent requests to remote storage.

Seer makes use of a simple algorithm to approximate the total shuffle data size D. Specifically, each mapper writes the size of its deserialized data partition to object storage, and the PyWren client retrieves and adds up all these values to compute the total shuffle data size D. Because the size of deserialized data is larger than the data in serialized form (recall that Seer utilizes Apache Arrow serialization), this approach overestimates the real data size, which may affect the accuracy of Seer. In our experiments, however, we did not appreciate a loss of prediction accuracy due to the small blowup factor across all workloads. In fact, the highest blowup factor showed up for the 100 GB TeraSort workload (§6), where the serialized and compressed shuffle write size was 79.5% lower than the data size in memory.

(Operators). Internally, we have used Pandas[5] to implement the operators (e.g., `groupBy`). In some cases, we modify Pandas internal implementation to improve performance. To wit, when the sorting key is smaller than the whole record, we manage the keys and records in a separate way as ⟨key, record-pointer⟩ tuples. We then sort only such tuples, thus better exploiting cache locality. We employ Pandas' views to avoid unneeded memory copies whenever possible.

(Stage identification). To support TPC-DS [28] queries in Seer, we have used the new output of the `explain` command available in Apache Spark 3. In particular, we have used the new mode called `formatted`, which returns the formatted physical plan of any query. The output not only includes the "naked" tree of the operators with a number in parenthesis that indicates their execution order, but the `id` of the `codegen` stage of each operator. With this knowledge, it is easy to: 1. Identify the stages and the tasks within a stage; and 2. Specify them using our annotation API. Stages are separated by `Exchange` operators (i.e., shuffle operations), while the tasks within a stage share the same `codegen id`.

5.1. Extensions

(Straggler mitigation). Similarly to other recent works [29,46], we have observed unpredictable performance variations in our job executions for large parallelism due to sudden slow access to object storage from some workers. To handle unforeseen variability, we have implemented a "lightweight" version of Late [43] that exploits speculative execution to eliminate network stragglers. We adopted Late, since FaaS platforms permit the quick launching of speculative replicas of the slowest workers without effort. In Seer, we speculate a percentage (by default, 10%) of the tasks.

(Pipelining). Seer supports the eager scheduling of reduce tasks in the second physical stage. The lazy approach simply starts up a reduce task only when all the map tasks have terminated, while the eager approach starts launching reduce tasks after 25% (default) of the map workers have finished. If the concurrency limit of $1,000$ concurrent invocations is hit, the second-stage workers are gradually spawned as soon as there are free slots. By default, Seer employs eager scheduling. But, it also supports the lazy approach, because each type of scheduling optimizes different key performance indicators. On the one hand, the lazy approach is cost-efficient [46]: since reduce tasks waste no time waiting for upstream map tasks to finalize, individual worker execution is minimized, and hence, the cost. On the other hand, the eager approach enables pipelining of the first and second physical stages, which minimizes the shuffle time, though at a higher cost.

6. Evaluation

In this section, we describe our evaluation plan. We first compare both shuffle methods and gauge the accuracy of our models (§6.1). Next, we assess our dynamic selection method using TPC-DS [28] (§6.2). To conclude, we evaluate Seer using the TeraSort benchmark (§6.3).

(Setup). Unless otherwise stated, the setup will be as follows. For the serverless workers, we use functions of 2 GB of RAM (i.e., 1 vCPU) and IBM COS for storage, all from the same region (`us-east`). We warm up the functions by default to alleviate the impact of the stragglers in our tests.

For our comparisons against Spark, we also use IBM Cloud in the `us-east` region and Apache Spark 3.2.0 (Oct 13, 2021). We create two clusters of different sizes. A larger cluster of 10 workers nodes for Terasort, and a smaller cluster of 4 nodes for TPC-DS. Worker nodes are instances of type `cx2-16x32` (16 vCPUs, 32 GB RAM). In both clusters, we set up a `bx2-2x4` instance (2 vCPUs, 4 GB RAM) to operate as master node. Network bandwidth (measured using Linux `iperf`[6]) was of 16.2 Gbit/s between worker nodes, and of 3.3 Gbit/s between the master and worker nodes. We deployed HDFS (Hadoop 3.3.1; replication level 1) in both clusters and used it to store the different datasets. We configured Spark to launch executors with 8 GB of RAM and 4 cores per executor. This gives us 2 GB of RAM per core on average, leading to an equivalent configuration with that of cloud functions. We enabled the Spark External Shuffle Service (ESS) for better fault tolerance.

For serverless Spark [13], Google Cloud allocates executors with the following configuration by default: 4 cores and $13,626$ MB of RAM, and leverages Spark's dynamic resource allocation mechanism to auto-scale them, starting at least from a minimum pool of two executors.

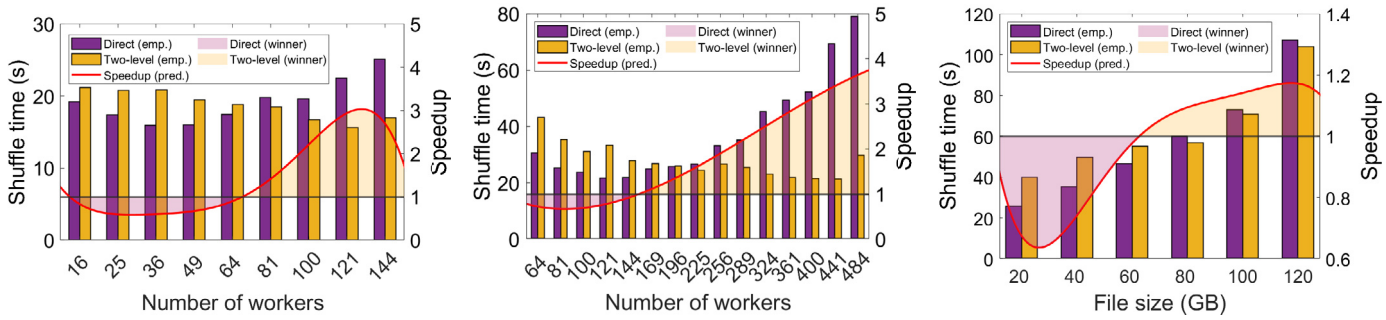
6.1. Direct or two-level shuffle?

Yet, one of the lingering questions in this work boils down to show that no shuffle implementation holds an edge over the other in all cases. We prove this in this section.

(Strong scaling). Strong scalability quantifies how well a shuffle method scales with increasing parallelism for a fixed workload. For this test, we use two real datasets from METASPACE [27] of 5.1 GBs (`CT26_Xenograft.csv`) and 19.7 GBs (`X089-Mousebrain_842x603.csv`), resp. We run a simple job composed of a `groupBy` operation. to group the x-coordinate of the associated mass spectrometry (MS) image [37].

Results are displayed in Fig. 9a for the Xenograft dataset and in Fig. 9b for the MouseBrain dataset. Non-surprisingly, the two-level method scales better with growing parallelism. However, such an observation overlooks the important subtlety that direct shuffle is on par with the two-level method. More concretely, direct shuffle delivers a comparative minimum latency to that of two-level shuffle, but with less amount of parallelism (e.g., about 3.6X less number of workers for MouseBrain).

[4] https://arrow.apache.org/docs/python/index.html.
[5] https://pandas.pydata.org/.
[6] iPerf, https://iperf.fr/.



(a) `Xenograft` **(Strong scaling)**. (b) `MouseBrain` **(Strong scaling)**. (c) `MouseBrain` **(Weak scaling)**.

Fig. 9. Strong and weak scaling of direct and two-level shuffle for two omics datasets.

Table 3
Start-up time for optimal parallelism in MouseBrain (19.7 GB).

Algorithm	Optimal # of workers	Shuffle time	Start-up time
Direct	121	21.78 s	7.30 s
Two-level	441	21.44 s	19.96 s

A smaller parallelism is highly desirable because it is less sensitive to stragglers, cold starts, etc. To better understand this, we report in Table 3 the start-up time for the optimal parallelism, or the time spent on starting up all the workers that minimizes the job completion time in the MouseBrain dataset. The table shows that the start-up time is much up higher for the two-level method, yet rendering equivalent shuffle time. So, direct shuffle is preferable in practice unless a higher degree of parallelism is needed for some reason.

(Weak scaling). We also measure the ability of each shuffle method to scale while keeping the problem size per worker constant. Cloud functions are hugely limited by memory (up to 2 GB for IBM Cloud Functions). It is thus relevant for many end users, e.g., those with restricted budget, to find what dataset sizes qualify as "shuffable" under the same resource usage per worker. For shuffle methods, this means observing what happens when the dataset size (D) and parallelism (p) increase in proportion. To investigate this, we kept the size of the input partitions fixed at approx. 25 MB, and increased parallelism so that $D = (0.0244) \times p$ gigabytes. Data for the input partitions came from the Mousebrain dataset.

Fig. 9c illustrates the weak scaling of both algorithms. The most relevant observation from this figure is that none of the two shuffle algorithms weakly scale. On one hand, direct shuffle is hindered by insufficient throughput. This occurs because if the amount of data processed per worker stays the same, the number of intermediate files grows quadratically as the dataset size increases. On the other hand, the two-level approach pays the price of passing two times the entire dataset. The $\mathcal{O}(\sqrt{p})$ reduction in the number of requests is unable to compensate the linear increase in the dataset size. This feature that is key to strongly scale renders useless to weakly scale. Yet this reinforces our expectations that no algorithm prevails under all scenarios.

(Accuracy). To assess the accuracy, all the subplots in Fig. 10 include the speedup (φ) curve of the two-level approach relative to direct shuffle as calculated by Algorithm 2. As shown in all subplots, our performance model is accurate, and characterizes pretty well the regions where one shuffle algorithm is better than the other. Some of the subfigures present a small underestimation of the true break-even point, with an average error of only 5.91%. This error is caused by overheads not captured by our model such as the variance of IBM COS, which is a globally shared service, or the effect of serialization on the estimation of the shuffle data size (§5).

6.2. TPC-DS

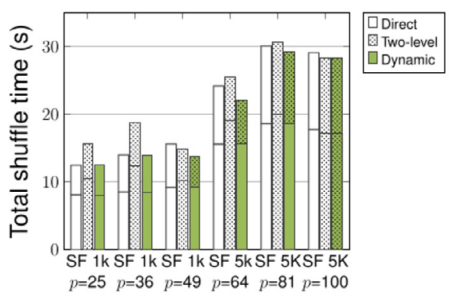
To confirm that Seer chooses the optimal algorithm for each stage of a real application DAG, we run Query 1 (Q1) from the TPC-DS benchmark [28]. This benchmark has a set of standard decision support queries typically used by retail product suppliers. Query 1 is very convenient, because its physical plan has a complex DAG comprising multiple stages that process a variable amount of data, which allow us to put under test the correctness of our shuffle manager. We evaluate Q1 at scale factors (SF) 1k and 5k, which represents a total input volume of 1 TB and 5 TB of data for various tables, respectively. For Q1, however, this amounts to 35 GB and 175 GB of input data in practice.

As detailed in §5, we extract the physical plan for query Q1 from Spark SQL, code it into Seer using our stage annotation API and execute it. This has the desirable side effect of enabling a **one-to-one** comparison between Seer and Spark, including the brand-new serverless Spark [13] service hosted in Google Cloud. In short, serverless Spark, which became generally available in early 2022, allows users to run Spark jobs without having to worry about right-sizing their cluster of virtual machines. In this sense, we wanted to see whether serverless Spark auto-scales well, and compared it to Seer (+ PyWren), a FaaS-based serverless solution.

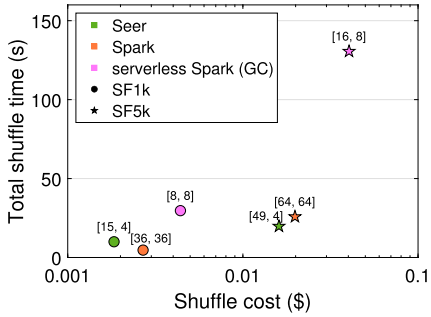
In addition to Spark, we also compared Seer to BigQuery [9], the Query-as-a-Service (QaaS) system from Google Cloud. All the tables per scale factor were exported to Google Cloud Storage (`us-east-1` region). Tables were loaded to BigQuery as not-partitioned external tables in CSV format. Table schemas were specified according to the official TPC-DS specs.

As our interest is only in shuffling through remote storage, we found it more appropriate to report the shuffle time of two distinct operations, rather than the total query time. Since the late stages of Q1 process much less data compared to early stages, we chose one early and one late operations in the query plan to see whether our system dynamically readjusts the number of workers. Specifically, we grabbed the `groupBy` operation on the `store_returns` fact table as an early operation. As a late operation, we chose one of the two `sort` operations from the final `SortMergeJoin` that implements the `orderBy` clause in Q1.

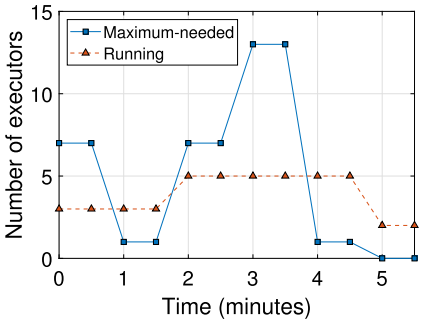
(Sanity Check). To spill clear water into the performance of Seer, we first conducted a sanity check to ascertain whether it correctly chooses the optimal algorithm as the degree of parallelism increases. To this end, we ran Q1 using three different configurations: one using our dynamic selection method (labeled '`Dynamic`' in Fig. 10a) against a version of Seer that always uses either the direct or two-level shuffle implementations (labeled '`Direct`' or '`Two-level`' in Fig. 10a).



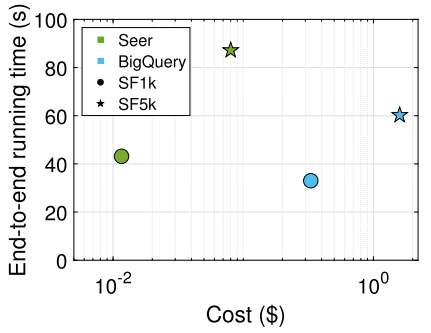
(a) **Seer, varying** p.



(b) **(serverless) Spark vs. Seer.**



(c) **Dynamic resource allocation in serverless Spark, SF 5k.**



(d) **BigQuery vs. Seer.**

Fig. 10. Results for TPC-DS Q1 workload for two scale factors 1k and 5k.

Results are illustrated as stacked bars. The upper bars report the shuffle time for the `sort` operator, whereas the lower bars do so for the heavier `groupBy` task. To aid interpretation, solid bars indicate the use of direct shuffle, whereas dashed bars pinpoint the use of the two-level algorithm. As shown in Fig. 10a, our results demonstrate that our dynamic approach always infers the optimal combination of shuffle algorithms for Q1 in both scale factors. In cases where both algorithms need to be mixed, our smart shuffle manager is able to capitalize on the optimal choice and diminish the overall shuffle time by 33.64% (SF 5k; 64 workers). As contrarily argued in [23], this figure proves that the two-level method is not always the best option.

(Spark). We compare Spark against Seer in two ways. On the one hand, we use our serverful deployment of Spark on IBM Cloud. On the other hand, we use the brand-new serverless Spark service from Google Cloud. For the former, we sought for the number of cores in Spark that delivered a performance **on par** with Seer, as Seer is faster than Spark with the same number of vCPUs (see §6.3). We needed to provision a cluster of 4 Spark servers. In the latter case, the TPC-DS tables were fetched from Google Cloud Storage and imported as Spark SQL tables.

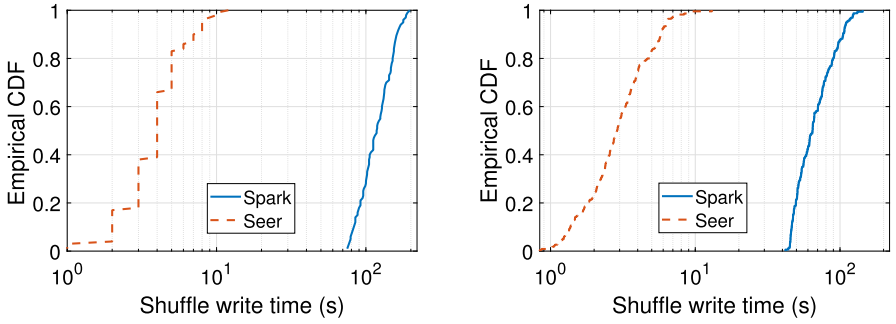
Results are provided in Fig. 10b. The numbers in brackets above the points indicate the amount of workers (or cores in Spark) that were allocated in the two shuffle operations. In terms of shuffle time, both Seer and Spark are the systems that exhibit the most constant latencies. While for Spark, this simply "ratifies" that the cluster had sufficient resources for both scale factors, this result verifies that Seer is able to use proportionally more workers as the data set grows. Actually, the Spark cluster was overprovisioned for the smaller scale factor. At SF 1k, Spark utilized only 36 vCPUs, half of the total compute power. Seer auto-adjusted the compute resources to the workload, achieving a similar performance to that of Spark with less resources for both scale factors (see Table 4). Although functions are more expensive than VMs per unit of time, Seer was between 24% – 46% cheaper than Spark thanks to its more fine-grained allocation of compute resources.

Table 4
Resource allocation breakdown at SF 5k.

System	Total # of workers	Total memory (GB)
Spark	$64 + 64 = 128$	$128 + 128 = 256$
Seer	$49 + 4 = 53$	$98 + 8 = 106$

In contrast to Seer, we discovered that serverless Spark does not scale up their resources as quickly as required to keep up with the current load. At SF 1k, it used only the two executors allocated by default, whereas for the largest scale factor, it strove to scale to four executors. Non-surprisingly then, Seer outperformed serverless Spark by 5.6X, yet being 1.5X less expensive. We were only able to scale serverless Spark to 9 executors (36 cores) by requesting them from the start. Fig. 10c depicts the number of **running** executors over time for the SF 5k, sampled every 30 seconds, as returned by the `Dataproc` service. In addition to the active number of executors, we also plot the curve for the **maximum** number of required executors. This metric is determined by Spark according to the length of pending task queue. The figure clearly shows that the number of active executors does not match at all the computing requirements of the workload. We speculate that this behavior is to avoid overprovisioning. In the end, perhaps, serverless Spark has not been conceived for short-running jobs, where serverless solutions like Seer can have the biggest advantage.

(BigQuery). As in the previous experiments, we ran Q1 in BigQuery. Since individual shuffle times are not available for this service, we only report in Fig. 10d the total query time. In terms of end-to-end execution time, BigQuery is somewhat faster than Seer at SF 1k. At SF 5k, the running time increases in both systems, though sublinearly, indicating that both systems use more resources for the larger scale factor. This is one of the main benefits of a shuffle manager such as Seer, where the number of workers can be automatically adjusted with the dataset size, en-



(a) **Spark(100w) and Seer(100w).** (b) **Spark(160w) and Seer(225w).**

Fig. 11. CDF of shuffle write time for a 100 GB TeraSort job. Numbers in parentheses indicate the # of workers.

suring sublinear query latencies. It must be noticed that, although the performance gap between both systems increases at SF 5k, it is difficult to tell why in a proprietary product such as BigQuery.

Further, Seer is significantly cheaper than BigQuery for both scale factors. The difference in both cases is about one order of magnitude. For instance, BigQuery is 19.8X more expensive that Seer for the large scale factor. This yields two interesting insights. First off, while a FaaS-based system like Seer runs on infrastructure that is rented per unit of time, the cost model of a QaaS system like BigQuery is based on the size of the data being stored and queried, which should be proportional to the resources used by the overall workload. However, it seems that for queries like Q1, a pricing model based on the number of bytes processed can be more costly than a one based on resource usage. Second, by jointly taking latency and cost, FaaS-based solutions like Seer can be more cost-efficient than QaaS systems for certain queries.

6.3. TeraSort

Most of the existing serverless analytics systems [29,36,46] make use of the TeraSort benchmark [42] to evaluate their performance. We see this workload as a good framework to compare Seer to state of the art. We sort 100 GB of data, since some of the systems (namely, Locus [29] and Caerus [46]) have not released their source code yet, but luckily, have performance numbers for that specific size in their respective articles. Further, we succeeded in executing the Terasort job in Primula [36], and ran it for serverful Spark for 100 and 160 cores, respectively.

To make a fairer comparison between serverful Spark and the serverless systems, we adjusted the number of input partitions to match the same number of cores in Spark, to avoid multiple rounds of `map` tasks. By default, Spark creates one input partition for each block of a file stored in HDFS. Since the default block size in HDF is of 128 MB, this would mean 800 `map` tasks, and between $5-8$ rounds of `map` tasks, which would be unfair. The same was done for the number of `reduce` tasks by adjusting the property `spark.sql.shuffle.partitions` to the number of available cores.

In this case, we measured the job execution time to make a fair comparison with the rest of systems. This time includes start-up time, the time to read the input data partitions from IBM COS by the `map` tasks and the time to write the output by the `reduce` workers. We used various scales of parallelism to provide a broader picture of its performance. For Spark, we used HDFS to store the Terasort dataset, which gave Spark a certain advantage over the serverless systems due to the exploitation of data locality.

Table 5 lists the execution time of the various approaches. In all cases, Seer outperforms the existing alternatives, with the exception of Locus for only 9 seconds, which is surprising, because Locus has been "bolstered up" with in-memory Redis instances. Further, we achieve a speedup of 2.17X compared with Primula [36], utilizing the exact same resource configuration for workers, i.e., 2 GB RAM workers. Interestingly, these results also confirm that the two-level method is heavily penalized by bandwidth: since each round writes and reads 100 GB of data, per-round transfer times cancel out the reduction in the number of requests. We notice that no configuration favors two-level shuffle in Table 5.

Table 5
Comparison of Seer to state-of-the-art data analytics systems for 100 GB TeraSort.

System		# workers	Storage layer	Exec. time
Qubole [30]		400	AWS S3	597.7 s
Locus [29]		dynamic	AWS S3/Redis	80 s to 140 s
Primula [36]		200	IBM COS	192.3 s
Caerus [46]		100	Jiffy [19] (VMs)	105 s
Serveful Spark		100	HDFS	600.12 s
		160		493.94 s
Seer	(+direct)	100	IBM COS	95.06 s
		225		89.96 s
		256		96.13 s
	(+two-level)	100	IBM COS	115.59 s
		225		100.91 s
		256		96.67 s

Compared with Spark, Seer is significantly better, around 6.31X faster for a hundred workers. Taking the best setup in each system, this gap reduces to 5.49X, which is equally high. Since this large difference in time can be caused by many reasons, such as the scheduling overhead [25], we decided to investigate the root cause. After careful inspection of the Spark log files, the cause for such an event turned up. Mostly, the large execution time in Spark was attributable to the high disk contention caused by the shuffle write operations. Recall that a shuffle write is ran independently for each map data partition that has to be shuffled, which may translate into concurrent random I/O writes to disk. Fig. 11 shows the CDF of the shuffle write times for both Seer and Spark. As seen in the figure, shuffle write times in Spark are close to two orders of magnitude higher than Seer, which accesses serverless object storage without noticeable contention, despite being disk-based storage too and thus, equally fault-tolerant.

(Comparison to Caerus [46]). We find specially interesting our comparison to Caerus. Caerus adopted Jiffy [19] in-memory data store for storing intermediate data. Since Jiffy runs on large VM instances (i.e., six `m4.16xlarge` EC2 instances), it was unexpected that Seer renders similar performance to that of Caerus using globally shared object storage. Caerus studied the impact of scheduling approaches on performance, specifically the lazy and eager scheduling of reducers, and proposed a new one: Nimble. Table 6 reports the execution time of the 100 GB TeraSort for these scheduling algorithms. Eager scheduling achieves better performance than lazy scheduling. Our most performant configuration in Seer is direct shuffle+ eager scheduling, achieving 1.13X lower job execution time compared to Nimble. This result shows that ob-

Table 6
Comparison with Caerus for TeraSort.

Scheme	Caerus			Seer +direct		Seer +two-level	
	Lazy	Eager	Nimble	Lazy	Eager	Lazy	Eager
Exec. time	124 s	105 s	107 s	105.7 s	95.06 s	125.7 s	115.59 s

Table 7
Pricing from IBM Cloud (`us-east`, Sept. 2022) and AWS (`us-east`, Mar. 2023).

Cost	Description	IBM Cloud Value	AWS Value
c_f	Function execution	1.7×10^{-5} ($/sec/GB)	1.7×10^{-5} ($/sec/GB)
c_i	Function invocation	0.0 ($/invocation)	0.2×10^{-6} ($/invocation)
c_s	Storage capacity usage	8.1×10^{-9} ($/sec/GB)	8.9×10^{-9} ($/sec/GB)
c_r	Read request	5×10^{-7} ($/op)	4×10^{-7} ($/op)
c_w	Write request	5×10^{-6} ($/op)	5×10^{-6} ($/op)

ject storage is a practical approach for many serverless analytics systems.

(Comparison to SONIC [22]). SONIC studied the best way to share intermediate data between functions: more concretely, by sharing files within a virtual machine (VM), copying them across VMs, or via object storage. We replicate one of their experiments to see if Seer is competitive against another smart management layer such as SONIC. To this end, we chose the MapReduce sort experiment with the same input volume of 1.5 GB. We evaluated both exchange algorithms equipped with lazy and eager scheduling schemes, as we did with Caerus. As a metric, we used Performance/$, defined as:

$$\text{Perf}/\$:= \frac{1}{\text{Execution time(s)}} \times \frac{1}{\text{Cost(\$)}}, \tag{3}$$

so that any improvement in latency, cost, or both, caused by our operators was subsumed into a single unit. We also employed the raw job execution time as a secondary metric to dissociate the $-cost normalization effect. **For Perf/S, higher is better**.

Importantly, we also ran Seer in the AWS cloud for this experiment. Our conviction was to evaluate whether the improvements introduced by Seer hold across different clouds. To make a fair comparison, we selected the memory allocation that had the equivalent of 1 vCPU in both cloud providers. As of today, this corresponds to 2 GB of memory for IBM Cloud Functions and 1,769 MB for AWS Lambda.

Because the Perf/$ metric requires a precise measurement of cost (see Eq. (3)), we developed a cost model to capture all the potential cost factors of a data shuffle within a single region. In a nutshell, the cost of a data shuffle operation consists of two parts: one for the compute functions and another for object storage. The computing cost comprises only two quantities: the duration cost of functions (c_f) measured as $/sec/GB of memory, and the cost of function invocations (c_i) measured as $/1M invocations. Additional chargeable features such as provisioned concurrency or extra ephemeral storage capacity in AWS Lambda[7] are not used by Seer.

The storage cost is in turn split into two classes of components: one for the storage capacity used as c_s $/sec/GB, and another for the operational requests charged as $/op. In some hyper-scalers (e.g., IBM Cloud), this cost breaks down further depending on the type of operation. As a result, we distinguished between the cost of read requests (c_r) from that of write requests (c_w), all measured as $/op.

Assuming that job execution happens in the same region, so there are no extra costs due to outbound data transfers, the overall cost is given by:

$$\text{Cost}_{method} := \underbrace{\left[c_f \times T_{method} + c_i\right] \times p}_{\text{Compute cost}} + \underbrace{(c_r + c_w) \times \frac{R_{method}}{2} + c_s \times T_{method} \times D_{method}}_{\text{Storage cost}}, \tag{4}$$

where T_{method} denotes the total shuffle time, R_{method} denotes the total number of storage requests and D_{method} represents the amount of intermediate data written to object storage. We have used the subscript "*method*" in all terms to indicate that these quantities depend on the specific shuffle method (see Table 1 for further details). For example, remember that the amount of intermediate data written to object storage is two times larger in the two-level method than in direct shuffle, i.e., $D_{2level} \approx 2 \times D_{direct}$.

We list the pricing information for IBM Cloud and AWS in Table 7. As shown in the table, storage costs are at least one order of magnitude smaller than compute costs, and might be thus ignored in practice. This large difference in cost is because object stores have been traditionally designed to persist data in the long term, and not to operate as a substrate for ephemeral data, thereby offering cheap prices to customers. On the contrary, functions have taken the opposite tack.

Fig. 12 shows the results for 30 workers as in SONIC. For two-level shuffle, we scaled down the number of workers to the nearest quadratic integer, namely 25. Additionally, Fig. 12 reports the optimal parallelism for each shuffle method derived from our performance model for the IBM Cloud, which was of 35 workers for direct shuffle and of 64 for two-level shuffle. For AWS, we ran Seer with the same set of configurations as for IBM in order to better cross-compare the improvements of our novel shuffle manager.

As seen in this figure, our implementation achieves up to 9.93X better Perf/$ than SONIC as a consequence of the higher gains in execution time, concretely up to 7.2X faster, delivered by direct shuffle with 35 workers (optimal configuration for the IBM Cloud). We note that the cost of functions is the dominating cost factor as listed in Table 7. Since such a cost is proportional to the shuffle time, the significant reduction in the duration of serverless functions accomplished by Seer also signified a steep decrease in the overall cost, leading to higher Perf/$ values. Also interestingly, the performance of Seer in AWS was equally good, which demonstrates that the core ideas behind Seer are general and straightforward to implement in other cloud providers as well. Irrespective of the shuffle method, we observe that eager scheduling achieved the best latency and Perf/$ results, as it allowed pipelining shuffle writes with shuffle reads.

(Summary of insights). Our comparison with state of the art leads to three key insights. First, one insight that holds across all the above scenarios is that direct shuffle is often the optimal solution

[7] https://aws.amazon.com/lambda/pricing/.

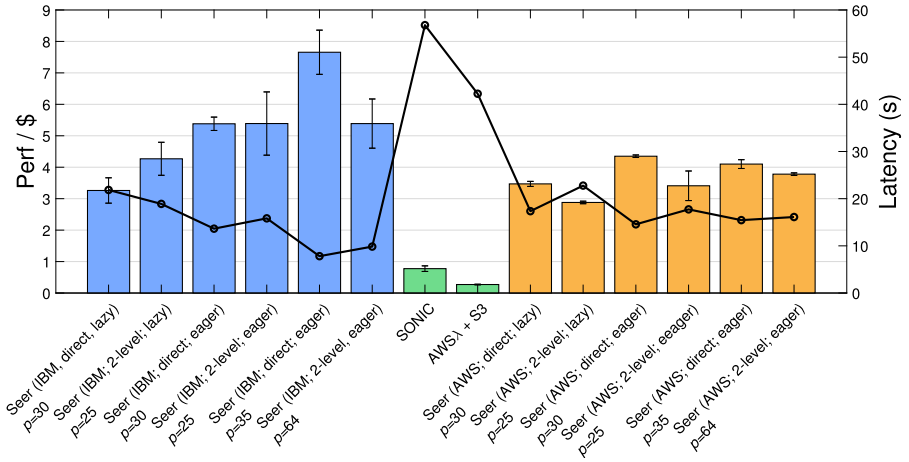


Fig. 12. Comparison with SONIC for MapReduce Sort. Bars represent Perf/$ (left axis). Points report execution time (right axis).

for practical problems, despite its quadratic request complexity. Second, sharing ephemeral data via object storage can be more cost-optimal than do so directly between VMs (as in SONIC) or through a storage cluster of VMs (as in Caerus), which makes object storage a solid substrate for serverless analytics. And finally, the core ideas behind Seer are general enough to get large improvements in other cloud providers, as shown in the SONIC experiments for IBM Cloud and AWS.

7. Related work

We are not the first to identify the problem of data passing as a key challenge for serverless analytics jobs [15,10,11,14,17,20,1,6,22]. Pocket [21] develops a multi-tier storage approach to improve the performance of ephemeral data sharing. gg [11] has shown that serverless functions can communicate directly using NAT traversal techniques in the AWS Lambda service. However, this requires that both endpoint functions are up at the time of the data transfer, which complicates fault tolerance. Object storage enables time decoupling instead, which favors resource elasticity: new functions can be launched without disrupting the others. Also, it enables the persistence of intermediate state, as data analytics frameworks such as Spark perform with local disks, which gives comparable fault tolerance guarantees.

Only a few works have specifically studied the problem of how to efficiently shuffle data in serverless architectures [29,23,36]. Locus [29] designs a multi-round shuffle algorithm that uses the direct approach to do smaller shuffles (e.g., of 100 GB). Locus adopts a hybrid architecture, combining object storage with fast, in-memory storage. The major drawback of this approach is that the in-memory instances have to be manually provisioned by users, which affects the usability of the whole system. To curtail the throughput demands, Lambada [23] proposed the two-level approach, but without comparing it thoroughly against the direct approach. As we have proven in this research work, the two-level approach is not a silver bullet, which calls for smart managers such as Seer.

Another related work is SONIC [22]. SONIC reduces job latency by jointly optimizing how data is shared —e.g., by replicating files across VM instances or sharing them through object storage. Unfortunately, SONIC leaves significant headroom for improvement since it does not optimize object storage data passing as we realize in Seer. Much in the same way, we see that a large collection of serverless data analytics systems (e.g., PyWren [16], Ripple [18], Caerus [46] and Kappa [47], etc.), may adopt Seer, or benefit from the insights contributed by this work. Seer only uses object storage and cloud functions, which are serverless services available in all public clouds. This makes Seer a very practical and portable solution across all cloud providers.

Adding a modicum of statefulness to serverless workflows, other works recognize the need for sharing state [6,40,8,35,19]. Crucial [6] improves data sharing and coordination by means of distributed shared objects. Cloudburst [40] focuses on distributed consistency of shared data through the Anna key-value store [41]. Cirrus [8] builds a parameter server abstraction on top of VM instances, which serves as the storage access point of the global model shared by the all functions. Further, Mlless [35] uses Redis instances to quickly exchange model updates between workers. Finally, Jiffy [19], an evolution of Pocket, realizes resource allocation at the granularity of memory blocks to keep up with the high elasticity of functions. These systems significantly improve data passing latency, but do not abide by a "pure" serverless architecture. Users of Crucial and Cirrus must provision the total number of nodes required in advance, while Jiffy and CloudBurst scale up their capacity by adding more servers into the data plane, which may take too much time for moderate data shuffles. For instance, Cloudburst is bottlenecked by the latency of spinning up EC2 instances (around 2.5 minutes), while 100 GB of data can be shuffled within 1.5 minutes via remote object storage.

8. Conclusion

We have presented Seer, a shuffle manager for serverless analytics that dynamically chooses the shuffle method, as well as the optimal degree of parallelism that maximizes the I/O efficiency to object storage. Seer resolves this decision problem formulating an analytical model, which has the desirable property that end users do not need to specify (or even estimate) intermediate data sizes at the job submission time, making Seer fully "serverless" by design. We show that for different analytics workloads, Seer is able to improve execution time by $1.1 - 7.2$X for the same number of vCPUs, which contributes to confirm that object storage is a practical solution to serverless shuffling.

Future work includes the exploitation of intra-level parallelism. Although IBM Cloud Functions can get as much as 1 vCPU, other FaaS platforms have access to more vCPUs (e.g., AWS Lambda have access to up to 6 vCPUs). This will enable "fatter" workers and the exploitation of data locality during shuffling, which has not been leveraged at all by Seer, and can save significant network bandwidth.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work has been partially funded by the Horizon Europe programme under grant agreements no. 101092644 (NearData), no. 101092646 (CloudSkin), and no. 101093110 (EXTRACT), as well as by the Spanish Government (no. PID2019-106774RB-C22). Germán T. Eizaguirre is recipient of a pre-doctoral FPU grant from the Spanish Ministry of Universities (ref. FPU21/00630). Marc Sánchez-Artigas is a Serra Húnter Fellow.

References

[1] I.E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, V. Hilt, SAND: towards high-performance serverless computing, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), USENIX Association, Boston, MA, 2018, pp. 923–935, https://www.usenix.org/conference/atc18/presentation/akkus.

[2] A. Arjona, P.G. López, J. Sampé, A. Slominski, L. Villard, Triggerflow: trigger-based orchestration of serverless workflows, Future Gener. Comput. Syst. 124 (2021) 215–229, https://doi.org/10.1016/j.future.2021.06.004.

[3] M. Armbrust, R.S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley, X. Meng, T. Kaftan, M.J. Franklin, A. Ghodsi, M. Zaharia, Spark sql: relational data processing in spark, in: 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15), ACM, 2015, pp. 1383–1394.

[4] AWS, S3 request limits, https://aws.amazon.com/premiumsupport/knowledge-center/s3-request-limit-avoid-throttling/, may 2021.

[5] Azure, Azure blob storage request limits, https://docs.microsoft.com/en-us/azure/storage/common/scalability-targets-standard-account, 2022.

[6] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, P. García-López, On the faas track: building stateful distributed applications with serverless architectures, in: 20th International Middleware Conference (Middleware'19), ACM, 2019, pp. 41–54.

[7] D.M. Beazley, Python gets an event loop (again), login Usenix Mag. 39 (3).

[8] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, R. Katz, Cirrus: a serverless framework for end-to-end ml workflows, in: ACM Symposium on Cloud Computing (SoCC'19), ACM, 2019, pp. 13–24.

[9] S. Fernandes, J. Bernardino, What is bigquery?, in: 19th International Database Engineering & Applications Symposium (IDEAS'15), ACM, 2015, pp. 202–203.

[10] S. Fouladi, R.S. Wahby, B. Shacklett, K.V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, K. Winstein, 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, 2017, pp. 363–376, https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[11] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, K. Winstein, From laptop to lambda: outsourcing everyday jobs to thousands of transient functional containers, in: 2019 USENIX Annual Technical Conference (ATC'19), USENIX Association, Renton, WA, 2019, pp. 475–488, https://www.usenix.org/conference/atc19/presentation/fouladi.

[12] Google, Google cloud storage request limits, https://cloud.google.com/storage/docs/request-rate, 2022.

[13] Google, Spark on Google cloud, https://cloud.google.com/solutions/spark, apr. 2022.

[14] J.M. Hellerstein, J.M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu, Serverless computing: one step forward, two steps back, in: 9th Biennial Conference on Innovative Data Systems Research (CIDR 2019), 2019, http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

[15] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Serverless computation with openlambda, in: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16), USENIX Association, 2016, https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[16] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht, Occupy the cloud: distributed computing for the 99%, in: 2017 ACM Symposium on Cloud Computing (SoCC'17), ACM, 2017, pp. 445–451.

[17] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J.E. Gonzalez, R.A. Popa, I. Stoica, D.A. Patterson, Cloud programming simplified: a Berkeley view on serverless computing, arXiv:1902.03383, https://arxiv.org/abs/1902.03383, 2019.

[18] S. Joyner, M. MacCoss, C. Delimitrou, H. Weatherspoon, Ripple: a practical declarative programming framework for serverless compute, arXiv:2001.00222, https://arxiv.org/abs/2001.00222, 2020.

[19] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, I. Stoica, Jiffy: elastic far-memory for stateful serverless analytics, in: Seventeenth European Conference on Computer Systems (EuroSys '22), ACM, 2022, pp. 697–713.

[20] Y. Kim, J. Lin, Serverless data analytics with flint, in: 11th IEEE International Conference on Cloud Computing (CLOUD'18), IEEE Computer Society, 2018, pp. 451–455, https://doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00063.

[21] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, C. Kozyrakis, Pocket: elastic ephemeral storage for serverless analytics, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association, Carlsbad, CA, 2018, pp. 427–444, https://www.usenix.org/conference/osdi18/presentation/klimovic.

[22] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, S. Bagchi, SONIC: application-aware data passing for chained serverless applications, in: 2021 USENIX Annual Technical Conference (USENIX ATC 21), USENIX Association, 2021, pp. 285–301, https://www.usenix.org/conference/atc21/presentation/mahgoub.

[23] I. Müller, R. Marroquín, G. Alonso, Lambada: interactive data analytics on cold data using serverless cloud infrastructure, in: 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), ACM, 2020, pp. 115–130.

[24] T. Neumann, Efficiently compiling efficient query plans for modern hardware, Proc. VLDB Endow. 4 (9) (2011) 539–550, https://doi.org/10.14778/2002938.2002940.

[25] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica, Sparrow: distributed, low latency scheduling, in: Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13), ACM, 2013, pp. 69–84.

[26] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, Making sense of performance in data analytics frameworks, in: 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15), USENIX Association, Oakland, CA, 2015, pp. 293–307, https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout.

[27] A. Palmer, P. Phapale, I.e.a. Chernyavsky, Fdr-controlled metabolite annotation for high-resolution imaging mass spectrometry, Nat. Methods 14 (2017) 57–60, https://metaspace2020.eu.

[28] M. Pöss, B. Smith, L. Kollár, P. Åke Larson, Tpc-ds, taking decision support benchmarking to the next level, in: 20th ACM SIGMOD International Conference on Management of Data (SIGMOD'02), ACM, 2002, pp. 582–587.

[29] Q. Pu, S. Venkataraman, I. Stoica, Shuffling, fast and slow: scalable analytics on serverless infrastructure, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19), USENIX Association, 2019, pp. 193–206, https://www.usenix.org/conference/nsdi19/presentation/pu.

[30] Qubole, Spark on lambda, https://www.qubole.com/blog/spark-on-aws-lambda/, may 2017.

[31] J. Sampé, G. Vernik, M. Sánchez-Artigas, P. García-López, Serverless data analytics in the ibm cloud, in: 19th ACM/IFIP Middleware Conference Industry (Middleware'18), ACM, 2018, pp. 1–7.

[32] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, A. Arjona, Toward multicloud access transparency in serverless computing, IEEE Softw. 38 (1) (2021) 68–74.

[33] J. Sampé, M. Sánchez-Artigas, G. Vernik, I. Yehekzel, P. García-López, Outsourcing data processing jobs with lithops, IEEE Trans. Cloud Comput. 11 (1) (2023) 1026–1037, https://doi.org/10.1109/TCC.2021.3129000.

[34] M. Sánchez-Artigas, G.T. Eizaguirre, A seer knows best: optimized object storage shuffling for serverless analytics, in: 23rd International Middleware Conference (Middleware'22), ACM, 2022, pp. 148–160.

[35] M. Sánchez-Artigas, P.G. Sarroca, Experience paper: towards enhancing cost efficiency in serverless machine learning training, in: 22nd International Middleware Conference (Middleware'21), ACM, 2021, pp. 210–222.

[36] M. Sánchez-Artigas, G.T. Eizaguirre, G. Vernik, L. Stuart, P. García-López, Primula: a practical shuffle/sort operator for serverless computing, in: 21st International Middleware Conference Industrial Track (Middleware'20), ACM, 2020, pp. 31–37, https://doi.org/10.1145/3429357.3430522.

[37] T. Schramm, Z. Hester, I. Klinkert, J.-P. Both, R.M. Heeren, A. Brunelle, O. Laprévote, N. Desbenoit, M.-F. Robbe, M. Stoeckli, B. Spengler, A. Römpp, imzml — a common data format for the flexible exchange and processing of mass spectrometry imaging data, J. Proteomics 75 (16) (2012) 5106–5110.

[38] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, S. Venkataraman, Serverless linear algebra, in: 11th ACM Symposium on Cloud Computing (SoCC'20), ACM, 2020, pp. 281–295.

[39] M. Shen, Y. Zhou, C. Singh Magnet, Push-based shuffle service for large-scale data processing, Proc. VLDB Endow. 13 (12) (2020) 3382–3395, https://doi.org/10.14778/3415478.3415558.

[40] V. Sreekanti, C. Wu, X.C. Lin, J. Schleier-Smith, J.E. Gonzalez, J.M. Hellerstein, A. Tumanov, Cloudburst: stateful functions-as-a-service, Proc. VLDB Endow. 13 (12) (2020) 2438–2452, https://doi.org/10.14778/3407790.3407836.

[41] C. Wu, J. Faleiro, Y. Lin, J. Hellerstein, Anna: a kvs for any scale, in: 2018 IEEE 34th International Conference on Data Engineering (ICDE'18), IEEE Computer Society, 2018, pp. 401–412.
[42] O.O. Yahoo! Terabyte sort on apache hadoop.
[43] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, I. Stoica, Improving mapreduce performance in heterogeneous environments, in: 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 08), USENIX Association, USA, 2008, pp. 29–42.
[44] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12), USENIX Association, San Jose, CA, 2012, pp. 15–28, https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.
[45] H. Zhang, B. Cho, E. Seyfe, A. Ching, M.J. Freedman, Riffle: optimized shuffle service for large-scale data analytics, in: Thirteenth EuroSys Conference (EuroSys '18), ACM, 2018.
[46] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, I. Stoica, Caerus: NIMBLE task scheduling for serverless analytics, in: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), USENIX Association, 2021, pp. 653–669, https://www.usenix.org/conference/nsdi21/presentation/zhang-hong.
[47] W. Zhang, V. Fang, A. Panda, S. Shenker Kappa, A programming framework for serverless computing, in: 11th ACM Symposium on Cloud Computing (SoCC'20), ACM, 2020, pp. 328–343.
[48] P. Zuk, K. Rzadca, Reducing response latency of composite functions-as-a-service through scheduling, J. Parallel Distrib. Comput. 167 (2022) 18–30, https://doi.org/10.1016/j.jpdc.2022.04.011, https://www.sciencedirect.com/science/article/pii/S0743731522000909.

Germán T. Eizaguirre received the B.Sc. degrees in Computer Engineering and Biotechnology, in 2020, and a M.Sc. degree in Computational Engineering and Mathematics, in 2021, from Universitat Rovira i Virgili (URV). He works since 2019 in Cloudlab, URV's cloud computing research group, where he has been developing his doctoral thesis since 2021. He has also collaborated with the Computational Biology and Systems Biomedicine Research Group of the Biodonostia HRI.

Marc Sánchez-Artigas received his Ph.D. degree in Computer Science in 2009 from the Universitat Pompeu Fabra, Spain. During his Ph.D. studies, he worked at Ecole Polytechnique Fédérale de Lausanne (EPFL). In the same year, he joined the Universitat Rovira i Virgili, where he currently works as an Associate Professor. He received the Best Paper Award from IEEE LCN'07 and the Best Dataset Award from ACM IMC'15. He has published 80+ articles in important venues such as IEEE P2P, ACM/IFIP Middleware, IEEE ICDCS, IEEE INFOCOM and USENIX FAST, among others. Now, he is coordinating the Horizon Europe project CloudSkin (grant no. 101092646), as well as actively being involved in the coordination of other Spanish and European projects in Cloud and Edge Computing.