Primula: a Practical Shuffle/Sort Operator for Serverless Computing

Marc Sánchez-Artigas
Universitat Rovira i Virgili
Tarragona, Spain
marc.sanchez@urv.cat

Germán T. Eizaguirre
Universitat Rovira i Virgili
Tarragona, Spain
germantelmo.eizaguirre@urv.cat

Gil Vernik
IBM Research
Haifa, Israel
gilv@il.ibm.com

Lachlan Stuart
EMBL
Heidelberg, Germany
lachlan.stuart@embl.de

Pedro García-López*
IBM Watson Research
New York, USA
pedro.garcia.lopez@ibm.com

Abstract

Serverless computing has recently gained much attention as a feasible alternative to always-on IaaS for data processing. However, existing severless frameworks are not (yet) usable enough to reach out to a large number of users. To wit, they still require developers to specify the number of serverless functions for a simple sort job. We report our experience in designing Primula, a serverless sort operator that abstracts away users from the complexities of resource provisioning, skewed data and stragglers, yielding the most accessible sort primitive to date. Our evaluation on the IBM Cloud platform demonstrates the usability of Primula without abandoning performance (e.g., 3x faster than a serverless Spark backend and 62% slower than a hybrid serverless/IaaS solution).

CCS Concepts: • **Theory of computation** → **Distributed computing models**; • **Computer systems organization** → **Cloud computing**.

Keywords: serverless computing, function-as-a-service

ACM Reference Format:
Marc Sánchez-Artigas, Germán T. Eizaguirre, Gil Vernik, Lachlan Stuart, and Pedro García-López. 2020. Primula: a Practical Shuffle/Sort Operator for Serverless Computing. In *1st International Middleware Conference Industrial Track (Middleware '20 Industrial Track), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3429357.3430522

*Also with Universitat Rovira i Virgili.

1 Introduction

Since their inception, serverless functions have received a huge amount of attention. One of the major reasons behind this increasing popularity is that serverless functions allow to get an extremely high degree of parallelism within a very short amount of time (typically a few seconds) and are billed at subsecond granularity. Non-surprisingly researchers soon realized that serverless platforms had the potential to make massive parallelization available to the masses [8, 9, 11, 18].

Despite these efforts, existing serverless frameworks are still not programmable enough for general users to express certain applications at a high-level without worrying about details like the optimal scale of parallelism, or the number of VM instances where to run non-serverless components. This is clearly visible for analytics workloads that need to move large data between functions that do not overlap in time. A illustrative example of this is sorting, which has at its core a shuffle operation to exchange data across all compute units. As direct transfer between functions is not allowed, existing serverless solutions have worked around this limitation by persisting intermediate data across stages on shared storage [13, 14, 16]. Despite important performance gains, most of these solutions require provisioning VM instances [13, 16], and thus cannot be qualified as serverless, or if they use only serverless components, they ask users to set up the number of serverless functions a job is split across [14], thus hurting performance if the scale of parallelism is not well chosen.

For novice users who seek to execute their single-machine code at scale, the above solutions are not yet practical. Too many impactful decisions have to be made beforehand, such as the type and number of VM or service instances [13, 16], or which configurations (on the pareto-optimal front) to pick [14], etc. Certainly, the pursuit of a higher level of simplicity is what motivated this work. Actually, such a need was first diagnosed by data scientists at the European Laboratory of Molecular Biology (EMBL) when they tried to implement a molecular annotation pipeline over IBM's general-purpose serverless framework named PyWren-IBM [18]. Concretely, mass spectrometry datasets needed to be sorted. However, the data scientists at EMBL found no high-level primitive in the existing literature to do so. Details on this use case of the EU H2020 CloudButton project [1] can be found in [3].

Contribution. We present the design of Primula, a serverless shuffle operator for general-purpose serverless frameworks. Among other features, Primula automatically infers the right scale of parallelism to optimize performance, handles skewed data through data sampling, and eagerly detects stragglers. We detail how we implement a high-level, practical sort primitive that needs no resource configuration parameters (e.g., number of functions, number of storage servers), thus achieving the premise of serverless computing of allowing developers to focus exclusively on the code. Actually, our integration with PyWren-IBM [18][1] requires developers a call to `sort()` with no parameters other than the location of the dataset, and the sort column. To our knowledge, this is the most accessible serverless sort primitive in the literature. Our results show that Primula predicts the optimal scale of parallelism pretty well and effectively mitigates stragglers.

2 Related work

Although numerous works have focused their attention on how to unleash the power of serverless computing for data analytics [6, 8, 9, 11, 18], only a few of them have tackled the problem of designing an efficient shuffle & sort operator for this brand new computing model. One of the first works was Flint [12], which proposed a rewrite of the Apache Spark [23] execution layer to run the tasks in a particular plan on AWS Lambda, instead of on a cluster. To shuffle intermediate data, however, Flint resorted to AWS SQS, which is slow and does not scale well [6]. To rapidly share ephemeral data, Klimovic et al. [13] designed an elastic, fully-managed storage system for serverless systems called Pocket. Although Pocket was used to run a 100GB sort with nice performance, the major issue with Pocket is that it runs in VMs, so starting up a new node can take hundreds of seconds. Clearly, this limitation compels users to rent the right number of VM instances in advance, which is what we avoid in Primula.

Most related to our work are Locus [16] and Lambada [14]. Locus designs a scalable shuffle operator that combines fast (AWS ElasticCache) and slow storage (AWS S3) to achieve a good trade-off between cost and performance. Similar to us, it includes an analytical model to estimate the right amount of resources. However, as in Pocket, storage resources need to be provisioned in advance, which jeopardizes the benefits of a serverless system, among other differences.

Like Primula, Lambada [14] is a pure serverless solution. Compared to Primula, it lacks of automated provisioning of workers, so the users are left with this non-trivial decision. Further, its greater performance comes from specific"tricks", which are not generalizable to all cloud platforms, such as to spread requests over multiple buckets to elude throttling due

[1]Now re-branded into Lithops [17] with many more features

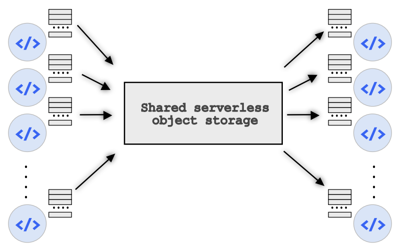


Figure 1. Architecture overview of Primula.

to a rate limit on requests. For instance, we applied this trick on the IBM Cloud Object Storage (COS) service and received only a marginal gain in performance. However, we view its multi-level exchange operator as a valuable asset to explore promptly for its notable reduction in the shuffle complexity.

3 Design

As a required building block for implementing sorting and other operations, the basic component of Primula is a shuffle operator for serverless workers. Since serverless functions cannot accept inbound connections, Primula employs object storage for inter-function communication, thus providing a *serverless-only* solution. To take a step further the serverless computing premise of simplifying resource provisioning, it estimates the number of workers that minimizes execution time (Sec. 4). Fig. 1 shows an overview of the system.

3.1 Basic algorithm

For Primula, we built a single-level shuffle operator, leaving the exploration of a multi-level approach like [14] for future work. While the single-level approach has the well-studied inconvenience that the total number of intermediate files is quadratic in the number of workers p (each of the p workers reads from and writes to p files), we needed first to ponder to what extent it was possible to automate sorting with good performance. According to this approach, we structured the shuffle operator into two stages that exchange data through object storage. In the first stage, each of the p workers holds its share of the input, partitions it, and writes the contents of each intermediate partition to object storage using a file naming convention that reflects the ID of the "receiver". In the second stage, each worker reads all files with its own ID. Using this scheme, we devised a basic sort as follows:

- In the first (or map) stage, each worker reads its own input partition in chunks of fixed size (Sec. 3.2) using a dedicated thread, while sorting each of them locally in another thread, so that the partial sorts can be masked by the I/O latency in reading from external storage. Finally, the partially sorted partitions are written back to object storage.
- In the second (or reduce) stage, each worker fetches its own intermediate data partitions, as determined by its

own ID, performs a final sort, and outputs the final result to object storage.

In the following sections, we detail the set of enhancements we added to this basic algorithm to improve execution times.

3.2 I/O granularity

As the size of the (intermediate) data partitions can very large (e.g., of 1 GB) for big data sets, it is not a good idea to use the data partition size as the default granularity for I/O requests. The main reason is that object storage systems, in contrast to distributed file systems like GFS [10] and HDFS [19], do not distribute objects across all servers, but manage them in their entirety. For large objects, this coarser granularity reduces the throughput of the object storage system drastically [5]. Indeed, as long as the file exceeds a certain size, the available network bandwidth becomes the limiting factor [5]. To help to hit a higher aggregate throughput, Primula accesses data partitions in fixed size chunks, instead of by means of a single whole-object request. The chunk size is configurable, as the optimal choice depends on the environment where Primula runs. For instance, we experimented with different sizes and found that the optimal value for the IBM Cloud is $\approx$ 64MB. It is expected that the optimal chunk size is of a few megabytes in practice. The majority of large scale storage systems have been optimized for reading and writing large chunks of data, rather than for high-throughput, small-sized operations.

3.3 Barrierless execution

The parallel implementation of sorting requires transferring data between its two stages. Primula implements behind the scenes the most commonly used communication pattern to do so, i.e., the shuffle pattern. In classical implementations of MapReduce, the two stages of the sorting operation would be separated by a barrier to ensure that all the relevant data is available to the second stage before it starts. In practice, the barrier is useful to atomically operate on all the records of a particular key in the `reduce()` function. However, there are some applications where this requirement is not longer necessary [21]. One of those practical use cases is sorting, where this relaxation can lead to significant improvement.

To further speed up execution, we therefore implemented a barrierless version of sorting. Compared with prior works such as [21], we use a simple online method where we launch all the second-stage workers after $x\%$ (default value is 20%) of the first-stage tasks have finished. If the concurrency limit has been reached (typically of $1,000$ concurrent invocations), second-stage workers are incrementally spawned as soon as the occupied slots by the first-stage workers are released.

3.4 Straggler mitigation

Similar to what was observed in [16], we found the presence of stragglers when sorting some datasets. A high percentage of them were caused by slow access to object storage, which was expected, given the aggressive I/O requirements posed by sorting/shuffling over a serverless environment. Among the slew of mitigation methods [4, 7, 24], we implement a lighter version of LATE [24] for all the sorting stages. This was made possible thanks to a fixed I/O granularity and the inherent elasticity of serverless architectures. On one hand, a predefined I/O size makes it easy to monitor the progress of each individual worker by taking as the progress score the fraction of input data processed. On the other hand, the high elasticity of serverless platforms enables the agile launching of speculative copies of the straggling workers.

Specifically, speculative execution in Primula is as follows: Each worker i periodically publishes its progress score ps_i (a value between 0 and 1) along with its running time T_i to the object store. A monitoring task, also running as a function, regularly fetches these values and estimates the completion time of worker i as: $\ell_i = (1 - ps_i) \times \left(\frac{ps_i}{T_i}\right)$. If the threshold on the number of speculative copies has not still been reached, the monitoring function first ranks the workers that have not been yet speculated by the estimated time left ℓ_i. Finally, it launches a copy of the highest-ranked task whose progress rate $\frac{ps_i}{T_i}$ is below a given threshold. In particular, we express this "slow-worker" threshold as an xth percentile of overall worker progress (default value is 25). We adopted this strategy because for jobs such as sorting where the progress rate is linearly related to actual progress, it only executes speculatively tasks that will improve the overall job response time, rather than just slow tasks. To prevent misestimation of the time left ℓ_i due to the cold start of some workers, we only consider those workers whose running time is 10% larger than the execution time of the last started worker.

In terms of implementation, we would like to stress that speculative execution has not been exclusively designed for sorting. On the contrary, it is general enough to seamlessly augment other serverless programming constructs, e.g., the `map()` primitive of PyWren [11, 18].

3.5 Data sampling

An inherent problem in big data workloads is data skewness. For sorting, data skewness is particularly sensible, as it will cause shuffle skew, where some second-stage workers will receive more data than the others. A trivial solution would be to allow users to enter the partitioning ranges manually to achieve perfect load balancing, but this would contradict the simplicity of the solution.

To automatically cope with data skewness, we develop a random sampling method similar to that available in Hadoop (`RandomSampler` [22]) to generate equally-sized partitions. This process is realized before the first stage of sorting, can be deactivated, and the result is a set of key intervals, which are used to indicate the boundaries of the intermediate data partitions. Unless specified otherwise, Primula samples 0.01% of the total dataset size.

To make data sampling more efficient, we implement it as a MapReduce job. Concretely, each of the p mappers reads $\frac{0.01}{p}$% of the dataset size and extracts its list of key intervals. These lists are finally merged on the reducer side to produce accurate boundaries for the intermediate data partitions.

4 Automated Scale of Parallelism

Given a data set size, Primula is able to resolve on the fly the optimal number of workers that minimizes the sorting time. As the memory size of functions is small, e.g., a maximum of 2048MB per function in the IBM Cloud, sorting becomes an I/O-bound task in practice. For the most part, this converts the problem of minimizing the sorting time into finding the number of workers that lead to a high utilization of storage resources, and more concretely, of object storage. We chose blob storage to store intermediate data because it is available as a "serverless " service in all popular cloud platforms. This design enables users to sort data on-demand, automatically scaled and billed only for the time that sorting runs, thereby fulfilling the promise of serverless computing.

To automatically infer the optimal scale of parallelism for each input size, we first develop a system model that allows us to compare different configurations and pick the best one. Like in [16], we made an effort to keep this model as much generic as possible to abstract away the singularities of each cloud provider. Since the goal is not to accurately predict the sorting time, but the optimal level of concurrency, the model does not account for details that do not straightaway affect parallelism, e.g., the sampling process to deliver correct load balancing between workers. Obviously, this simplification underestimates the real sorting time. But it does not prevent the model from inferring the ideal number of workers.

4.1 Model

We assume that the volume of data to be sorted is of D bytes. The degree of parallelism represents the number of *workers* or cloud functions that execute in parallel, which we denote by p. Since the amount of data to be processed in each phase remains constant, no matter the dataset size, the number of workers needed in each phase will always be p. This means that each worker will require at least $\frac{D}{p}$ bytes of memory to store its partition.

As discussed above, Primula permits users to control the granularity of data transfers to achieve optimal performance. This implies that the workers read from and write to object storage in chunks of c bytes. This also offers the side benefit of making it possible to reduce execution time by pipelining I/O activity with useful work. Without chunking, it is trivial to see that the total number of intermediate data partitions would be p^2, that is, each of the p workers would operate on p intermediate data partitions. With chunking, the number of requests, however, grows to $\left\lceil \frac{D}{c \times p^2} \right\rceil p^2$ requests. While this apparently puts a higher strain on the object storage system, the scaling factor $\left\lceil \frac{D}{c \times p^2} \right\rceil$ typically becomes a small constant for practical values of c, so the number of requests is $O(p^2)$.

Similar to [16], we consider that the object storage system can limit the performance of sorting due to two reasons: 1) a limit on *bandwidth* per worker; and 2) a scarce operation *throughput*. Since bandwidth can be asymmetric, we denote the bandwidth available per individual worker as b_r and b_w bytes/sec for reads and writes, respectively. We also assume that the object storage system limits the aggregate number of read and write operations/sec. We denote these limits as q_r and q_w, respectively.

Now denote by $T_{\text{sort}}(p)$ our prediction of the sorting time. As the sorting operation is divided into two stages, we have

$$T_{\text{sort}}(p) = T_{\text{map}}(p) + T_{\text{rdc}}(p), \tag{1}$$

where $T_{\text{map}}(p)$ and $T_{\text{rdc}}(p)$ denote the time it takes to fulfill the map and reduce stages of the sort, respectively.

In the shuffle stage, each worker reads its input partition of size $\frac{D}{p}$ bytes and sorts it locally in a pipelining way, finally writing the intermediate sorted subpartitions back to object storage. Clearly, the writing phase of this stage is the most I/O-intensive. It requires to perform $\left\lceil \frac{D}{c \times p^2} \right\rceil p^2$ write requests with an overall throughput of q_w writes/sec. Assuming that q_w is the bottleneck, the time it takes to complete them is $T_{q_w}(p) = \frac{\left(\left\lceil \frac{D}{c \times p^2} \right\rceil p^2\right)}{q_w}$. Likewise, assuming a per-worker write bandwidth limit of b_w bytes/sec, this time becomes $T_{b_w}(p) = \frac{D}{b_w \times p}$, when b_w is the bottleneck[2]. Now accounting for both bottlenecks, the time to write all the intermediate partitions is thus $max\left\{T_{q_w}(p), T_{b_w}(p)\right\}$. For this stage, we get

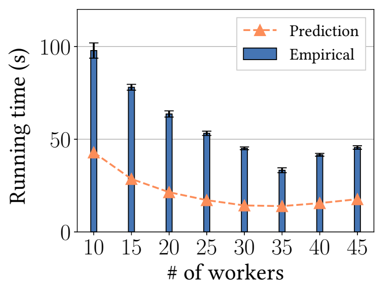
$$T_{\text{map}}(p) = \underbrace{\max\left\{\frac{D}{b_r \times p}, \frac{\left\lceil \frac{D}{c \times p} \right\rceil p}{q_r}\right\}}_{\textit{Reading of input share}} + \underbrace{\max\left\{\frac{D}{b_w \times p}, \frac{\left(\left\lceil \frac{D}{c \times p^2} \right\rceil p^2\right)}{q_w}\right\}}_{\textit{Writing of intermediate data}}.$$

In the second stage, each worker reads all its corresponding intermediate partitions and merges them. Finally, the sorted data is output to object storage. By an analogous reasoning to above, it is not difficult to see that the reduce phase of the sort thus takes

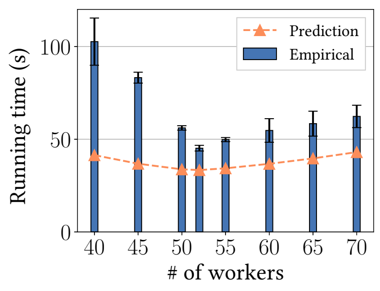
$$T_{\text{rdc}}(p) = \underbrace{\max\left\{\frac{D}{b_r \times p}, \frac{\left(\left\lceil \frac{D}{c \times p^2} \right\rceil p^2\right)}{q_r}\right\}}_{\textit{Reading of intermediate data}} + \underbrace{\max\left\{\frac{D}{b_w \times p}, \frac{\left\lceil \frac{D}{c \times p} \right\rceil p}{q_w}\right\}}_{\textit{Output of sorted data}}.$$

Although (1) is not always differentiable at the minimum for particular choices of c, the search domain is discrete and bounded: $\left\lceil \frac{D}{w} \right\rceil \leq p \leq \mathcal{P}$, where w is the memory capacity in bytes allocated to the workers, and $\mathcal{P}$ denotes the maximum

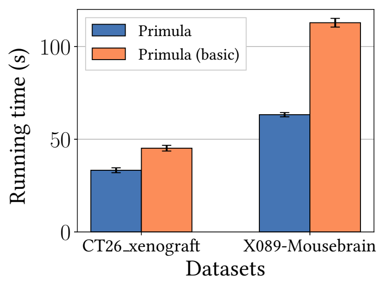
[2]Note that the expression for $T_{b_w}(p)$ assumes that the bandwidth provided by the object storage service scales linearly as more workers are added. We verified this to be true for IBM COS for the maximum concurrency limit of 1,000 workers.



(a) Scale of parallelism predicted by Primula versus actual measurements for `CT26_xenograft`.



(b) Scale of parallelism predicted by Primula versus measurements for `X089-Mousebrain_842x603`.



(c) Performance of Primula against a basic version w/o enhancements.

Figure 2. Performance and model accuracy of Primula for two different metobolomics datasets.

allowed concurrency limit (e.g., of 1,000 on IBM Cloud and AWS, by default). This ensures that the global minimum can be found after a few evaluations of Eq. (1). Results on the accuracy of this model can be found in the evaluation (Sec. 6).

5 Implementation

Primula has been written in approximately 1, 960 lines of code by extending PyWren-IBM [18] (now Lithops [17]), a serverless-based MapReduce engine maintained by IBM. Although PyWren-IBM allows to easily implement many data-parallel tasks, it lacks of an actual shuffle operator and sort primitive. Thus, we have augmented it with support for both shuffle and sort operations, implementing all the features described before. To be non-intrusive, we did not modify the original PyWren-IBM stack, inheriting its original overheads (e.g, the cost of dynamically injecting code to workers), which causes extra roundtrips to the object storage and hurts performance. No need to mention that we used IBM Cloud Functions for the workers and IBM COS as the serverless storage system.

As the in-memory sorting algorithm, we use TimSort [15], which performs very well for data that is partially ordered. To better exploit cache locality when they sort key is smaller than the entire record (e.g., sort key occupies 10 bytes while the whole record is 100 bytes), we have augmented each key with a pointer to the record, managing the keys and records separately. This way, each sort comparison only required a cache lookup that was mostly sequential, instead of a slow, random access to memory. Internally, Primula uses Pandas DataFrames API to operate on and share intermediate data among the workers.

To conclude, we include a small snippet of code to show how easy it is to sort with Primula a CSV file:

```python
import pywren_ibm_cloud as pywren

pw = pywren.ibm_cf_executor()
pw.sort("cos://us-east/german-data/Mouse_brain.
    csv",primary_sort_column=1)
```

Listing 1. Sort example.

6 Evaluation

We evaluate Primula with two metabolomics datasets, and with a 100GB Terasort. Specifically, with our evaluation, we set out to provide evidence to the following questions: *What is the accuracy of the predictive model for the correct number of workers?*, *What is the impact of the enhancements over the basic algorithm?*, *Is it effective in mitigating stragglers?*, and finally, *What is its performance compared to prior work?*

Setup. Unless otherwise noted, we utilized workers with 2048 MB of main memory, and a single bucket in IBM COS for all storage operations. All the dataset files are not partitioned in advance. They are retrieved in parts by fetching byte-ranges in the map stage of sorting. All experiments were run on US East (`us-east`) region.

6.1 Model accuracy and enhancements

Recall that to automatically determine the optimal number of workers, Primula relies on the predictive model of Sec. 4. To gauge its accuracy, we used the two metabolomics datasets from the METASPACE platform [2] that sparked this work: `CT26_xenograft` and `X089-Mousebrain_842x603`, both in imzML format, which after their conversion to CSV format have a size of 5.1GB and 19.7GB, respectively. Concretely, we used Primula to sort both datasets with varying number of workers and compared the results with the predicted values.

As can be seen in Fig. 2(a) and Fig, 2(b), Primula's model predicts the scale of parallelism very well. It achieves perfect accuracy for the `X089-Mousebrain_842x603` dataset, where the optimal worker number is 52, and 2.94% relative error in the smaller dataset. Clearly, these figures show the ability of the model to identify the point beyond which the bottleneck shifts to IBM COS's throughput and execution time increases. Beyond that point, the utilization of more workers is harmful, incurring more cost while not improving performance. This is the key reason why the decision on the scale of parallelism should not be passed onto the user. Although not our focus, we notice that a useful prediction of the actual sort time is

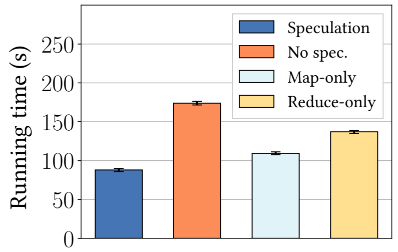


Figure 3. Running times with stragglers. *Map*- and *Reduce-only* refers to map- and reduce-only stragglers, and without speculation.

possible by accounting for the overheads not modeled by Eq. (1) such as data serializing/deserializing, etc.

Finally, Fig. 2(c) compares Primula's performance against the basic version described in Sec. 3.1, but without chunked access to data. The goal of this comparison was to measure the influence of the design optimizations (Sec. 3). We set the number of workers to the optimum for each dataset, i.e., to 34 (`CT26_xenograft`) and 52 (`X089-Mousebrain_842x603`), respectively. Clearly, the results shows that our extensions to basic algorithm can lead to significant improvements, e.g., of close to 2x in the bigger dataset, with the gap increasing with the dataset size, which verifies their combined positive effect in performance.

6.2 Straggler mitigation

To assess the efficacy of our straggler mitigation method, we manually slowed down three workers from a total of 30 by introducing delays of 4s to 10s in the I/O operations. We did so in both stages, or either in the map or reduce stages. For this test, we used a 10GB dataset generated by Teragen [22].

As can be seen in Fig. 3, the load was significant enough that disabling speculative tasks caused performance to drop by more than 2x, but not so significant as to utterly impede the completion of the sort operation.Concretely, on average, speculative execution finished the sort job 51.4% faster than without "fail-slow" fault mitigation enabled.

In the 100GB sort experiment of the next section, Primula mitigated two real stragglers on average, typically, arising in the map stage of sort. In one of the runs, we even observed that one of the stragglers would have led a map task to time out (i.e., to exceed the execution limit of 600s for workers) if it was not backed up by a speculative copy.

6.3 Performance comparison

We compare the performance of Primula with the numbers published for implementations [3] alike that use a single-level shuffle operator like ours, namely Locus [16] and Qubole [20].

[3]Lambada [14] is multi-level, and as of today, its source code is unavailable.

Table 1. Running time of sort operations.

	# of workers	storage layer	time
Qubole [20]	400	AWS S3	579.7s
Locus [16]	unreported	AWS S3/Redis	≈ 120s
Primula	200	IBM COS (1 bucket)	212.4s
		IBM COS (10 buckets)	196.2s
		IBM COS (25 buckets)	195.4s

Qubole is primarily a serverless backend for Spark [23], and it is a good example of what may happen when recasting an existing system to a new technology. It is very important to note that both systems utilize AWS Lambda for the workers and AWS S3 as external storage backend. This makes direct comparison difficult, since these services behave differently across cloud providers. However, we found this comparison interesting to determine the position of Primula in terms of performance. For this experiment, we ran a 100GB Terasort, and instrumented Primula to use 1, 10 or 25 buckets to verify whether it is possible to improve throughput by the usage of multiple buckets as observed in [14] for AWS S3.

Table 1 shows the running time of the three approaches. Compared to Qubole, Primula runs 2.96x faster with the half of workers, which suggests that refitting an existing system to a new technology could be counterproductive. Compared to Locus with 2GB-workers, Primula is approximately 62.3% slower than Locus, which is a pretty good result, assuming that Locus uses a cluster of r4.2xlarge EC2 instances (61GB memory, up to 10Gbps network) to run the Redis instances, whereas Primula leverages only serverless components. A final observation to be made is that the multi-bucket "trick" to bypass rate limits used in Lambada [14] is of little help in the IBM Cloud platform.

7 Conclusions

We have presented Primula, a usable serverless shuffle/sort operator that abstracts aways the complexities of resource provisioning from users. Primula automatically handles data skewness and mitigates stragglers, and is able to accurately determine the degree of paralellism needed for a sort job. We have finally shown that Primula performance is solid across different datasets, competitive to existing tools, yet keeping cloud management overhead close to zero. Our future work includes the extension of our shuffle operator to other data-parallel abstractions and applications.

Acknowledgments

Source code available at https://git.cloudlab.urv.cat/geizaguirre/primula. This work has been partially supported by the EU Horizon 2020 programme under grant agreement 825184 and by the Spanish Government (PID2019-106774RB-C22). Marc Sánchez-Artigas is a Serra Húnter Fellow.

References

[1] 2019. EU H2020 CloudButton. https://cloudbutton.eu/.

[2] 2020. EU H2020 METASPACE. https://metaspace2020.eu/.

[3] Theodore Alexandrov and Gil Vernik. 2020. Decoding Dark Molecular Matter in Spatial Metabolomics with IBM Cloud Functions. https://www.ibm.com/cloud/blog/decoding-dark-molecular-matter-in-spatial-metabolomics-with-ibm-cloud-functions.

[4] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *9th USENIX Conference on Operating Systems Design and Implementation(OSDI'10)*. 265–278.

[5] Ali Anwar, Yue Cheng, Aayush Gupta, and Ali R. Butt. 2016. MOS: Workload-Aware Elasticity for Cloud Object Stores. In *25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)*. 177–188.

[6] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *20th International Middleware Conference (Middleware'19)*. 41—-54.

[7] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04)- Volume 6*. 10.

[8] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 475–488.

[9] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. 363–376.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 29–43.

[11] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *2017 Symposium on Cloud Computing (SoCC'17)*. 445–451.

[12] Y. Kim and J. Lin. 2018. Serverless Data Analytics with Flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD'18)*. 451–455.

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

[14] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 115–130.

[15] Tim Peters. 2015. Timsort description. http://svn.python.org/projects/python/trunk/Objects/listsort.txt.

[16] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 193–206.

[17] Josep Sampé, Pedro García-López, Marc Sánchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. 2020. Towards Multicloud Access Transparency in Serverless Computing. *IEEE Software* (2020).

[18] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud. In *19th International Middleware Conference Industry (Middleware'18)*. 1–8.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. 1–10.

[20] Bharath Bhushan Venkat Sowrirajan and Mayank Ahuja. 2020. Qubole Announces Apache Spark on AWS Lambda. https://www.qubole.com/blog/spark-on-aws-lambda/.

[21] A. Verma, N. Zea, B. Cho, I. Gupta, and R. H. Campbell. 2010. Breaking the MapReduce Stage Barrier. In *2010 IEEE International Conference on Cluster Computing (Cluster'10)*. 235–244.

[22] Tom White. 2015. *Hadoop: The Definitive Guide, Fourth Edition* (4th ed.). O'Reilly Media, Inc.

[23] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. 15–28.

[24] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 29–42.