ETH Library

Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure

Conference Paper

Author(s):
Müller, Ingo; Marroquín, Renato; Alonso, Gustavo

Publication date:
2020-06

Permanent link:
https://doi.org/10.3929/ethz-b-000413183

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
https://doi.org/10.1145/3318464.3389758

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure

Ingo Müller
ingo.mueller@inf.ethz.ch
ETH Zurich

Renato Marroquín
marenato@inf.ethz.ch
ETH Zurich

Gustavo Alonso
alonso@inf.ethz.ch
ETH Zurich

ABSTRACT

Serverless computing has recently attracted a lot of attention from research and industry due to its promise of ultimate elasticity and operational simplicity. However, there is no consensus yet on whether or not the approach is suitable for data processing. In this paper, we present Lambada, a serverless distributed data processing framework designed to explore how to perform data analytics on serverless computing. In our analysis, supported with extensive experiments, we show in which scenarios serverless makes sense from an economic and performance perspective. We address several important technical questions that need to be solved to support data analytics and present examples from several domains where serverless offers a cost and performance advantage over existing solutions.

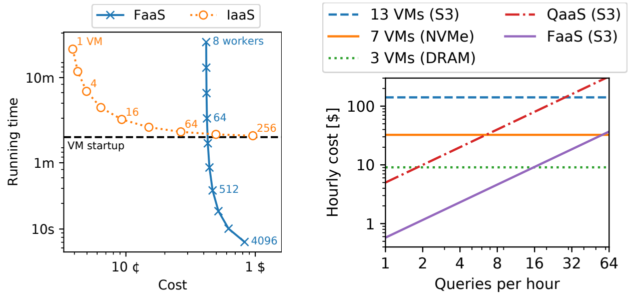
CCS CONCEPTS

• **Information systems → Parallel and distributed DBMSs**; *Online analytical processing engines*; *Database query processing*.

KEYWORDS

Serverless Computing; Serverless Functions; Cloud Computing; Interactive Analytics; Data Lake; Elasticity.

ACM Reference Format:
Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389758



(a) Job-scoped resources. **(b) Always-on resources.**

Figure 1: Comparison of cloud architectures.

1 INTRODUCTION

Data processing in the cloud has become a widespread solution in a wide variety of use cases. In the early days of Infrastructure-as-a-Service (IaaS), the cloud provided bare computing resources in the form of virtual machines (VMs). It then evolved into a richer computing and development experience through Platform-as-a-Service (PaaS). In both cases, the basic assumption was that the cloud is used as a rented computing infrastructure, whose elasticity can lead to a lower total cost than owned infrastructure. However, elasticity was limited by how fast the infrastructure could be started and stopped, and services migrated. Thus, cloud offerings evolved further towards Software-as-a-Service (SaaS), where customers rent the use of a certain software stack. Google BigQuery [20] or Amazon Athena [19] are examples of Query-as-a-Service (QaaS) systems providing database services without having to run (and pay for) a database server.

The demand for even higher elasticity and more fine-grained billing has recently led to the proliferation of Function-as-a-Service (FaaS). FaaS implements serverless computing—a name that emphasizes precisely the advantages of the approach: there is no need to install, operate, and manage a server (infrastructure) to get computations done. Applications such as source code compilation [11, 26] or video encoding [2, 11] have been shown to work well in such a setting.

To understand when FaaS is attractive for data analytics, consider a query scanning 1 TB of data stored on Amazon Simple Storage Service (S3). There are two ways to use IaaS for this task: starting a set of resources for the duration of a single

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389758

job (“job-scoped” resources) or scheduling jobs onto resources that are kept running (“always-on” resources). Figure 1a shows the costs and running time of job-scoped resources obtained through simulation for a varying number of workers.[1] As the plot shows, for both VMs and serverless functions, adding more resources reduces the running time, but with a diminishing gain as we approach the respective startup time.[2] To obtain the lowest cost, IaaS is thus more attractive, being up to an order of magnitude cheaper. However, if query latency matters, FaaS is more attractive. The strength of FaaS compared to job-scoped IaaS in data analytics is thus the ability to service *interactive* queries.

An alternative way to use IaaS is to keep resources running. This allows the system to load the data up-front and hence answer queries interactively as well. We thus extend our simulation to three systems loading the data into different levels of the memory hierarchy, which we assume can be read at full bandwidth. We choose the number of VMs such that the query above can be processed in 10 s: three large VMs for DRAM, seven for NVMe, and thirteen if we process the data directly from S3 without pre-loading.[3] Figure 1b shows the expected hourly cost of the different configurations as a function of the query frequency. Running virtual machines incurs only hourly costs, which is independent of the frequency at which queries are run. In contrast, FaaS and QaaS have a usage-based pricing model. Price increases linearly with the number of queries, such that even a moderate query load makes them more expensive than IaaS. The strength of FaaS compared to always-on IaaS is thus for *sporadic use*.

Combining the two arguments shows for which types of workloads serverless functions are most attractive for data analytics: *interactive* queries on *cold* (i.e., infrequently accessed) data. We refer to this use case as that of the “lone-wolf data scientist,” an individual or a small group of people during the interactive exploration of data sets that are otherwise rarely accessed, and give two concrete examples in the paper.

As promising as the idea might seem, using FaaS is controversial since serverless functions come with significant limitations: restricted network connectivity, limited running time, stateless operation with a very limited cache between invocations, and lack of control over the scheduling of functions. All of these shortcomings have been comprehensively analyzed in the literature [17, 24, 26, 27]. In the context of data analytics, the most severe limitation of FaaS is arguably the inability to have direct communication between function invocations. Previous work proposes a number of approaches, all of which involve running additional infrastructure on traditional VMs. Any such “serverful” component has the potential to severely limit the attractiveness of FaaS—as shown by the introductory example.

In this paper, we address the question of whether FaaS can be efficiently and effectively used for data analytics. Specifically, we present *Lambada*, a data analytics system on top of FaaS. We identify the technical limitations of FaaS and propose solutions that require *only serverless components*. One major novelty is an exchange operator that scales to several thousand workers *without* additional infrastructure—a problem that previous work could not solve [24, 26, 27, 38, 41]. Lambada is able to answer ad-hoc queries over gigabytes to terabytes of cold data at interactive query latency. In the most favorable cases, it is two orders of magnitude cheaper and one order of magnitude faster than commercial QaaS offerings. By building a full system, we show that the infeasibility conjectures of previous work [17] were not justified and that data analytics on FaaS is in fact technically possible and economically viable, albeit only for sporadic, interactive use.

In summary, the paper makes the following contributions:

- We characterize interactive analytics on cold data as the sweet spot for using FaaS for data analytics. This can be seen in part as a negative result—most other use cases seem to be handled well by traditional VMs.
- We present two use cases from scientific domains with requirements that match what FaaS can offer.
- We conduct extensive micro-benchmarks to determine the cost and performance characteristics as well as the limitations of current FaaS offerings.
- We implement a full-fledged data analytics system using only serverless components. Specifically, we propose solutions for efficient batch-start of massive numbers of serverless workers, a cloud-native scan operator for efficiently reading query input, and an exchange operator for inter-function communication.
- We compare the resulting performance and cost to those of other serverless solutions using end-to-end workloads from several domains and thus quantify the potential advantage of FaaS.

2 OVERVIEW OF LAMBADA

2.1 Suitable Cloud Infrastructure

For building a data analytics engine, serverless functions alone are not enough since they only execute code. However, we can use other cloud services to complement them. To preserve the advantages of FaaS, we should only use services with similar deployment and pricing models. In particular, these services should not incur any cost for idle infrastructure during think time or between usage sessions.

For compute, Amazon offers AWS Lambda, AWS Fargate, and Amazon EC2 to run code in a function, a container, and

[1] Between 1 and 256 `c5n.large` instances and between 8 and 4096 concurrent function invocations with 2 GiB main memory, respectively.

[2] We assume 2 min start-up time for IaaS and 4 s for FaaS.

[3] Our simulation uses `r5.12xlarge`, `i3.16xlarge`, and `c5n.18xlarge` instances, respectively.

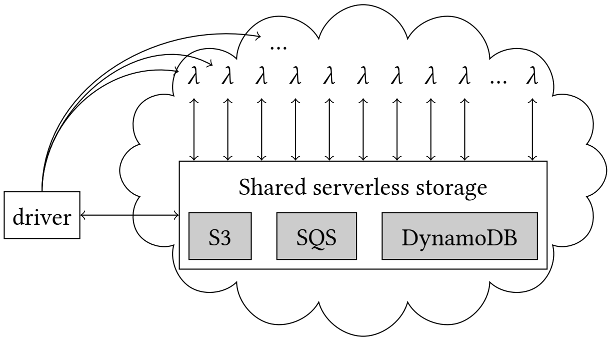


Figure 2: Architecture overview of Lambada.

a virtual machine, respectively. All of these could be used on a per-query basis and could thus qualify as pay-as-you-go, but, as we study in more detail below, only AWS Lambda has low enough start-up times for interactive analytics. For storage, Amazon offers DynamoDB and S3, which both scale to zero if used for temporary data during query execution. Finally, Amazon offers a message queue service (Amazon SQS) and a workflow service (AWS Step Functions), whose pure pay-per-use pricing model makes them suitable as well. Similar services can be found in the offerings of the other major cloud providers [6, 21].

In contrast, we argue that using any type of Platform-as-a-Service can compromise the attractiveness of a purely serverless system. For example, caching services such as Amazon ElastiCache run on (managed) virtual machines that are dedicated and paid for by the user, which has the same disadvantages as using VMs for compute.

2.2 Architecture Overview

For Lambada, our design goal is thus to use *solely* existing serverless components. Figure 2 depicts its high-level architecture. The *driver* runs on the local development machine of the data scientist. When she executes a query, the driver invokes a (potentially large) number of *serverless workers* (depicted as λ in the plot), who execute the query in a data-parallel manner. The workers communicate through different types of *shared serverless storage*: the cloud storage system *Amazon S3* for large amounts of data, the key-value store *Amazon DynamoDB* for small amounts of data, and the message service *Amazon SQS* (Simple Queuing System) for short messages. Input and output are read from and written to shared storage as well. In a way, this is a classical shared storage database architecture, except that *all* communication of the workers goes through shared storage and there is no direct communication between the workers or with the driver. The driver also uses the shared storage to communicate with the workers once they have been invoked, for example, to collect the results of their query fragments.

2.3 Data-parallel Query Plans

Queries are written in a thin Python front-end and go through a series of translations that transform it into an executable form. Our implementation is based on the Collection Virtual Machine (CVM) [34], a query compilation and execution framework that we are building in our group as part of a larger effort. A query plan in CVM is divided into *scopes*, each of which may run in a different target platform. Most operators in a typical plan of Lambada run in a serverless scope, i.e., are executed by the serverless workers. However, queries may also contain small scopes running on the driver, in order to do some pre-processing such as reading small amounts of data locally that should be broadcasted into the serverless workers or post-processing like aggregating the intermediate worker results.

2.4 Serverless Workers

The serverless workers run as a *function* in AWS Lambda, which is set up at installation time. Such a function consists of an event handler in one of the supported languages,[4] a "dependency layer" that may contain arbitrary native machine code, and some metadata such as the desired amount of main memory and the timeout of the function. The function of serverless workers consists of a dependency layer containing the same execution framework that also runs on the driver and an event handler as a wrapper around it implemented in Python. This event handler extracts the ID of the worker, the query plan fragment, and its input from the invocation parameters of the function and forwards them to the execution framework. It starts the execution engine in a new process with a memory limit slightly lower than that of the serverless function such that it can report out-of-memory situations and other errors of the execution engine to the driver rather than dying silently. When the execution engine finishes its computation, the handler forwards its results to the driver. In both cases, if an error occurred or the computation finished successfully, the handler posts a corresponding message into a *result queue* in SQS, from which the driver polls until it has heard back from all workers.

2.5 System Components for Serverless Analytics

While Lambada's architecture is very similar to a traditional shared storage database architecture, implementing such an architecture in a purely serverless environment is challenging. In the following sections, we identify a number of issues in the current serverless offerings and design system components that overcome all of them. Specifically, we propose solutions for efficient batch-start of massive numbers of serverless workers (Section 3), a cloud-native scan operator for efficiently reading query input (Section 4), and an S3-based exchange operator

[4] As of writing, AWS Lambda supports Node.js, Python, Java, Ruby, C#, Go, and PowerShell.

Metric	Region			
	eu	us	sa	ap
Single invocation time [ms]	36	363	474	536
Concurrent inv. rate [inv./s]	294	276	243	222
Intra-region rate [inv./s]	81	79	84	81

Table 1: Characteristics of function invocations.

that overcomes the scaling limitations of previous proposals (Section 5).

Each of the components needs to trade off three things: (1) hard quotas and limits from the service-level agreements (SLAs) of the cloud provider such as a limit on the request rate to S3, (2) execution speed under the given constraints (from service limits or from *de-facto* performance of a resource), for example, by overlapping communication with computation, and (3) usage-based cost of the various serverless services, such those from the running time of the serverless workers but also from the number of requests to the various systems.

3 BATCH-INVOCATION

3.1 Limits of Sequential Invocation

As a first component, we discuss how to invoke the serverless workers. Invoking a large number of them within a short amount of time is challenging. Table 1 shows the invocation characteristics of AWS Lambda functions in different regions from our location in Zurich, Switzerland. A single invocation takes between about 36 ms and 0.5 s, depending on the data center and our distance to it. By overlapping enough concurrent requests at the same time, we can largely hide the latency of the network round-trip: By using 128 threads to do the invocations, we achieve a rate of 220 invocations/s to 290 invocations/s for any of the data centers. However, this means that invoking 1000 workers from the driver still takes 3.4 s to 4.4 s and linearly more for more workers. With this approach, the invocation of the serverless workers can thus dominate the running time of the actual query.[5]

3.2 Lambada Two-level Invocation

To reduce the time until all serverless workers are invoked, we parallelize the invocation process by off-loading it partially to the first workers. More precisely, the workers that are invoked by the driver receive as additional parameter a list of IDs and input data. Before running their query fragment, each of this first generation of workers invokes a second-generation worker for each ID/input pair in that list. As serverless workers can invoke other workers at a rate that is in the same ballpark as that of the driver (see Table 1), a reasonable approach is to assign the same amount of invocations to the driver and each of the first-level workers, i.e., about $\sqrt{P}$ invocations each, where P is the total number of workers.

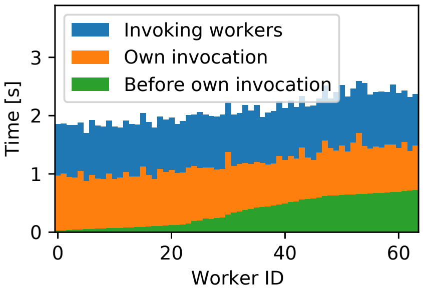


Figure 3: Example run of two-level invocation process.

Figure 3 shows the timings of an example run using this approach to start 4096 serverless workers based on a freshly created function (i.e., performing a cold start). It shows a timeline with three phases of every first-generation worker in the order they are invoked by the driver: (1) the time the driver took before it initiated their invocation (namely, to launch all previous workers), (2) the time their invocation took, i.e., the time between their invocation was initiated and they were actually running, and (3) the time they took to do the second-generation invocations. As the plot shows, the invocation of the last worker was initiated after about 2.5 s, which is tremendously faster than the 13 s to 18 s that the driver would be expected to take for doing the invocations alone based on the invocation rates from Table 1.

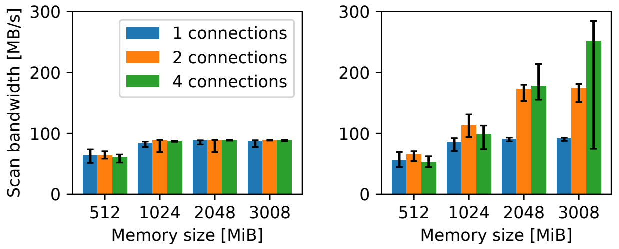
Note that the limit on the invocation rate of AWS Lambda is not relevant: it is currently ten times the limit on the number of concurrent invocations (i.e., workers) per second. Each query only needs *one* invocation per worker and the single user of our function is expected to run queries at a rate orders of magnitudes lower than ten per second. The limit on concurrent invocations, however, *is* relevant and we discuss some more details in Section 6.

4 CLOUD STORAGE SCAN OPERATOR

4.1 Network Characteristics

We first study the characteristics in terms of performance and cost of accessing S3 from the serverless workers in order to derive design principles for implementing scan operators. Figure 4 presents microbenchmarks for downloading large and small files from S3 into serverless workers using different configurations. We run each configuration three times in direct succession in nine different data centers using ten workers in each run. We compute the median, minimum, and maximum bandwidth of all workers in each data center and plot the medians of the three values as the colored bars, the lower error,

[5] If the query contains a synchronization point such that the workers need to wait for each other, then this also adds a quadratic component to the monetary cost.



(a) Large files (1 GB). **(b) Small files** (100 MB).

Figure 4: Network (ingress) bandwidth of serverless workers.

and the upper error, respectively. The plots thus show the distribution in a "median" data center.

For large files (Figure 4a), there is a very stable limit of about 90 MiB/s per worker.[6] Workers of virtually any size have fast enough network to achieve this limit; only workers with less than 1 GB of main memory see a slightly lower ingress bandwidth. Furthermore, using more network connections does not significantly change the overall bandwidth.

For small files (Figure 4b), the picture looks different. Workers with large amounts of memory observe a much higher network bandwidth, occasionally reaching almost 300 MiB/s. However, this is only the case if they use several network connections at the same time. We do not have access to information about Amazon's network infrastructure, but we assume that it uses a credit-based traffic shaping mechanism to limit the network bandwidth of each function instance to the 90 MiB/s observed above. Such a mechanism would allow bursts to exceed the target limit for a short amount of time and thus explain our results. In experiments not shown, we observe that the time span during which the burst may exceed the target is a small number of seconds. In order to maximize performance for short-running scans, we thus need to use multiple concurrent connections.

The fact that the memory size of the workers has an influence on the network bandwidth can be explained by the following: The cloud provider allocates an amount of CPU resources to each function that is proportional to its memory size.[7] More precisely, the allocation is such that a function with 1792 MiB gets the equivalent of one vCPU and functions with more memory get proportionally more. We can thus use a small amount of parallelism to overlap communication and computations and hide latencies with concurrent requests.

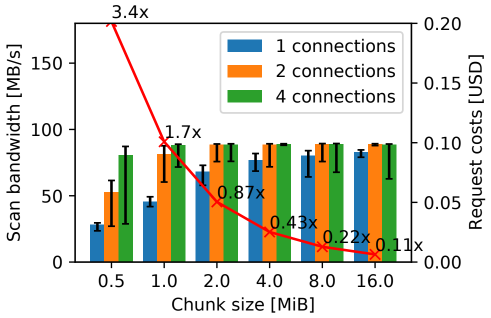


Figure 5: Impact of the chunk size on scan characteristics.

We also study the impact of the size of each individual request to S3 on the bandwidth and the cost of a scan. To that aim, we download a file of 1 GB with requests of different sizes using a variable number of connections. Figure 5 shows the result for the largest available serverless workers (i.e., with 3008 MiB of main memory). While a single connection requires a chunk size of 16 MB to get reasonably close to the maximum throughput from the previous experiment, we achieve that throughput even with a chunk size of 1 MB using four concurrent connections. This is the classical technique of hiding the latency of one or more requests with the processing of another. The size of each request also has a direct impact on the overall costs of a scan: it is inversely proportional to the *number* of requests, each of which has a fixed cost. One million read requests currently cost[8] $0.4. The line in Figure 5 shows the costs of running the experiment one thousand times. It is annotated with the factor by which the requests are more expensive than running the serverless workers. For example, in a scan with a chunk size of 1 MiB, the requests are 1.7 × more expensive than the workers' cost for the same scan. With even smaller chunk sizes, the requests can easily dominate the overall cost. In order to support small reads from S3, we thus need to support several in-flight requests but also avoid small reads wherever possible.

4.2 Lambada Cloud-native Scan

We use the above insights to design a scan operator that uses the network and CPU resources efficiently. We describe the design of a scan operator for Parquet files as an example, but expect the design of other operators to be conceptually similar. Parquet files are not only well-suited because they are widespread and optimized for slow storage, but they are also configurable in several ways such that they exhibit many characteristics that other formats might have.

[6] This is about 2 × higher than the numbers reported by Jonas et al. [24] published in 2017. It is also qualitatively different from the results of Wang et al. [43], who reported a stronger correlation between network bandwidth and worker memory size published in 2018. We assume that Amazon has increased the network bandwidth since then.

[7] See https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html

[8] In the "us-east-1" region, see https://aws.amazon.com/s3/pricing/#Request_pricing.

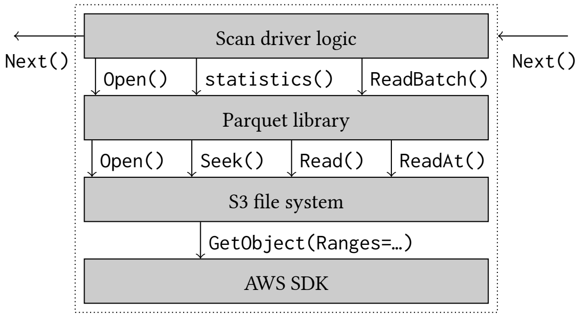


Figure 6: Components of the Parquet scan operator.

Figure 6 shows the main components of the operator. To the outside, it implements the open/next/close operator interface, through which it reads one or more file paths from its upstream operator and returns their content to its downstream operator as a sequence of table chunks in columnar format. In a typical plan, these chunks are consumed by a JiT-compiled pipeline, whose first operator is a scan operator for in-memory table chunks, which extracts individual records. Internally, the Parquet scan operator uses the official C++ library for Parquet files[9] to handle the deserialization of data and metadata. We have implemented the user-level filesystem interface of that library with a backend for S3, which, in turn, uses Amazon's AWS SDK for C++ to make requests to the S3 REST endpoint over the network.

The Parquet format has been designed to enable pushing down selections and projections. To that aim, the data is stored in consecutive groups of rows called *row groups*, each of which stores its records as consecutive columns called *column chunks*. Each column chunk may use a light-weight and a heavy-weight compression scheme, such as run-length encoding and GZIP, respectively. Furthermore, the footer of the file contains (optional) min/max statistics as well as absolute offsets for each column chunk. The library loads this metadata with a single file read, exposes the statistics to the scan operator such that it prunes out row groups based on its predicates, and loads the column chunks of the projected attributes when the scan operator accesses the remaining row groups using read operation per column chunk. Each of these read operations on the file system is translated to one request to S3, which downloads the desired bytes of the file (using HTTP's `Ranges` header). The file system offers a random-access interface (through `ReadAt`, as opposed to a stream-like interface through `Seek` and `Read`) which supports multiple concurrent reads.

We identify four levels where concurrent connections could be used to maximize bandwidth utilization on small files and small chunks, thus addressing the insights from Figures 4b and 5, respectively: (1) making several requests for each read operation in the filesystem, (2) downloading different column chunks of the same row group, (3) downloading multiple row groups at the same time, and (4) downloading data or metadata from different files at the same time. We always exploit level (4) by consuming the list of paths eagerly and downloading the metadata for all files that should be scanned in a dedicated thread in order to hide the latency of these small requests. Next, we exploit level (3) by downloading the data of up to two row groups asynchronously in two dedicated threads, except if the worker has too little main memory. This also overlaps the download(s) of one row group with the decompression and subsequent processing of the previous one. For small files and files with a single row group, we exploit level (2) by downloading different column chunks using multiple threads. We only fall back to level (1) if none of the other levels could be exploited as this would increase the number of requests and thus the costs of a scan. We expect that a similar prioritization to apply to other formats as well.

Finally, we exploit the (small amount of) multi-core parallelism in the workers with more than 1792 MiB of memory by optionally parallelizing the decompression of column chunks. This is only beneficial if decompressing a column chunk is slower than downloading it, which is only the case for the most heavy-weight compression schemes, and if the remaining query has too little compute to utilize the resources fully.

5 EXCHANGE OPERATOR

As one major building block for data processing, we design a family of exchange operators for serverless workers. Since serverless workers cannot accept incoming connections, they can only communicate through external storage. In order to support exchanging large amounts of data, we use S3 for this purpose.

5.1 Exchange in Joins, Sorting, and Grouping

The exchange operator[10] was introduced with the Volcano execution model [13, 14] to encapsulate any form of data parallelism and has since then become a central building block for data-parallel query processing [9, 15, 29, 36, 39]. In a data-parallel plan executed on a number of workers, the exchange operator transfers its input among the workers such that all tuples belonging to the same partition (according to some partitioning criteria) end up at the same worker. Joins, sorting, and grouping can be executed in parallel with the help of one or more exchange operators; no further operator with communication logic is required. For example, a parallel equi-join can be expressed by one exchange operator on each side of the input, which move all potential join partners to the same worker,

[9] The C++ library for Parquet is part of Apache Arrow, see https://github.com/apache/arrow.

[10] Alternative names include (re)partitioning, (re)distribution, shuffling, All-to-All (personalized) communication, or total exchange.

Algorithm 1 Basic S3-based exchange operator.

```
1: func BasicExchange($p$: Int, $\mathcal{P}$: Int[1..$P$], $R$: Record[1..$N$],
           FormatFileName: Int × Int → String)
2:     partitions ← DramPartitioning($R$, $\mathcal{P}$)
3:     for ⟨receiver, data⟩ in partitions do
4:         WriteFile(FormatFileName(receiver, $p$), data)
5:     for source in $\mathcal{P}$ do
6:         data ← data ∪ ReadFile(FormatFileName($p$, source))
7:     return data
```

followed by a local join operator. The operator we propose in this section is thus at the same time necessary and sufficient for data-parallel processing on serverless workers.

5.2 Basic Ideas and Challenges

Algorithm 1 shows how the basic exchange algorithm works, which other authors have used as well [24, 27, 38]: Each worker p of the P workers holds its share R of the input relation and uses an in-memory partitioning routine to split its input into P partitions, for example, based on the hash value of their key. It then writes the data of each partition into a file whose name reflects its own ID as well as the ID of the "receiver" of the file. Finally, it reads all files where its own ID has been used as the receiver sent by any of the other "source" workers. Since the sender may be slower than the receiver, the receiver must repeat reading a file until that file exists.

The problem with this algorithm is that the total number of files is quadratic in P: each of the P workers reads from and writes to P files. This may cause throttling by the cloud provider due to a rate limit on requests. For 1k workers, one execution of BasicExchange needs 2M requests while, as of July 2018, the rate limit on AWS is 3.5k and 5.5k per second for writes and reads, respectively,[11] and was as low as 300 and 800 read and write requests per second before that.[12] This effect has been pointed out by previous work [24, 27], which solved the problem by running their own storage service on rented VM instances.

There is another fundamental disadvantage with Algorithm 1 that will most likely not be solved by increased rate limits in the future and has not been mentioned by previous work so far: it incurs prohibitive costs, which also grow quadratically in the number of workers. As of writing, 1M read and write requests cost $5 and $0.4, respectively.[13] The left-most bars (labeled 1L) in Figure 7 show how the cost of BasicExchange evolves with the number of workers. With 256 workers, the costs for the requests to S3 are already higher than the costs for running the workers in most typical configurations, which are indicated by the horizontal range. With 4k workers, running the algorithm on 4 TiB costs about $100 for the requests to S3 and $3.3 for running the workers.

Algorithm 2 Two-level S3-based exchange operator.

```
func TwoLevelExchange($p$: int, $P$: int, $R$: Record [1..$N$])
    $\langle p_1, p_2 \rangle \leftarrow H_s(p)$
    $\mathcal{P}_i \leftarrow \{q | q \in \{1..P\} : q_i = p_i\}$ for $i = 1, 2$
    $f_i \leftarrow \langle s, t \rangle \mapsto$ "s3://b{$i$}/snd{$s$}/rcv{$r$}" for $i = 1, 2$
    tmp ← BasicGroupExchange($p$, $\mathcal{P}_1$, $f_1$, $R$, $H_s^2$)
    return BasicGroupExchange($p$, $\mathcal{P}_2$, $f_2$, tmp, $H_s^1$)
```

In the remainder of this section, we present two orthogonal optimizations that reduce the number of requests.

5.3 Lambada Multi-Level Exchange

The first optimization is to do the exchange through multiple levels, where each level only involves a small subset of the workers. This idea is a well-known technique in the parallel algorithms and HPC communities (see Grama et al. [16, Chapter 4.5] and references therein).

For two levels, we project the partition and worker IDs onto a grid and first do a horizontal exchange and then a vertical exchange. To that aim, we define the projection function $H_s := \langle H_s^1, H_s^2 \rangle := x \mapsto \langle x \,\%\, s, x \mathbin{/\!\!/} s \rangle$, which projects a number $x \in \{1..P\}$ onto two coordinates, where s is the desired number of distinct elements in the first dimension and % and $/\!\!/$ are modulo and integer division, respectively. Note that this approach works also for non-quadratic numbers of workers P. As a building block, we use BasicGroupExchange, which is the BasicExchange as defined before extended by a parameter for a projection function H_i, which it applies to the partition IDs while running the in-memory partitioning routine (Line 2 in Algorithm 1).

Algorithm 2 shows how the two-level exchange works. We first compute the two-dimensional worker ID from p. We then define the set of coworkers $\mathcal{P}_1$ that have the same value in the first coordinate and run BasicGroupExchange on the input to exchange data with this subset of workers. To do so, we parameterize the routine the following way: First, we use f_1 as FormatFileName, which prefixes all file names with a distinct bucket of this level, "s3://b{1}". Second, we have it apply the projection function H_s^2 to the partition ID, which means that it considers the second coordinate of the IDs for this round of the exchange. When this function returns, the second coordinate of the partition ID of any record coincides with that of the worker ID where it resides. Finally, we reverse the roles of the first and second coordinate and run BasicGroupExchange again, now among the group of workers induced by the other coordinate, $\mathcal{P}_2$. After this step, the first

[11] See https://aws.amazon.com/about-aws/whats-new/2018/07/amazon-s3-announces-increased-request-rate-performance/

[12] See https://forums.aws.amazon.com/message.jspa?messageID=573975#573975.

[13] In the "us-east-1" region, see https://aws.amazon.com/s3/pricing/#Request_pricing.

Table 2: Cost models of S3-based exchange algorithms.

Algorithm	#reads	#writes	#lists	#scans
1l	P^2	P^2	$O(P)$	1
1l-wc	P^2	P	$O(P)$	1
2l	$2P\sqrt{P}$	$2P\sqrt{P}$	$O(P)$	2
2l-wc	$2P\sqrt{P}$	$2P$	$O(P)$	2
3l	$3P\sqrt[3]{P}$	$3P\sqrt[3]{P}$	$O(P)$	3
3l-wc	$3P\sqrt[3]{P}$	$3P$	$O(P)$	3

coordinate of the partition ID of any record also coincides with that of the worker ID where it resides, so the exchange is complete.

The two-level approach reduces the number of requests, as the number of each phase grows only quadratically with the *group size* instead of the number of workers. More precisely, each worker does P/s read and write requests in the first level and, by definition, s in the second. Thus, together the P workers do P^2/s and Ps requests in the first and second level, respectively. It is easy to see that $s = \sqrt{P}$ minimizes the sum of the two terms, so we use this value for the rest of the paper. In total, the algorithm does hence $2P\sqrt{P}$ read and write requests each. At the same time, it reads and writes the input two times instead of just one, which increases run time and hence the cost of running the workers. We study this trade-off in more detail below.

The same idea can be applied to three or more levels to reduce the number of requests even further. For k levels, the partition and worker IDs are projected onto a k-dimensional grid with side length $\sqrt[k]{P}$ and BasicGroupExchange is used k times, once for each dimension (each of which reads and writes the input once). Table 2 summarizes the characteristics of the different algorithms.

5.4 Lambada Write Combining

The second optimization consists in writing all partitions produced by one worker into a single file. We call this technique "write combining." Instead of reading one entire file per sender, the receivers now need to read *part of one file* per sender. We thus define FormatFileName such that it ignores the parameter value for the receiver. We use Parquet as the format of the files in the exchange operator, so the senders simply write one row group per receiver. The receivers use the scan operator from Section 4 to read only the row group that they are responsible for.

5.5 Complexity and Cost Analysis

In Figure 7, we compare the costs of the different algorithms. Here, we show i exchange levels with and without write combining (wc). To compute the costs for the requests, we use the cost models from Table 2 at the rates quoted above ($5 and $0.4 for 1M read and write requests cost, respectively). The lower bars in full color represent the read cost, the upper bars in lighter color represent the write cost of the respective algorithms. BasicExchange is labeled 1l, while the two- and three-level variants are labeled 2l, and 3l; variants using write combining are suffixed with -wc. The horizontal range shows the costs of running the workers. For the purpose of this plot, we assume that they achieve 85 MiB/s, do not experience waiting time, and each second costs $\$3.3 \times 10^{-5}$ (which is the current price on AWS for workers with 2 GiB RAM). The lower edge of the range represents the running costs of the workers doing one scan on an input of 100 MiB while the upper edge represents the costs for three scans of 1 GiB. This range helps to put the costs of the requests into perspective.

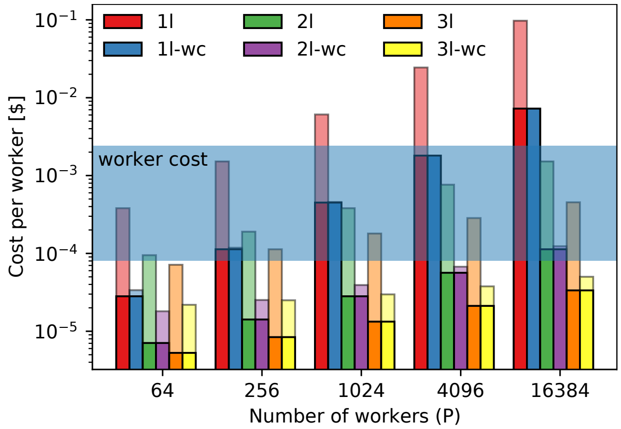


Figure 7: Cost of S3-based exchange algorithms on AWS.

As observed before, the plot shows that the costs of BasicExchange do not scale with the number of workers. While using write combining reduces the write costs to a negligible amount, the read costs, which still grow quadratically, can still be dominant in many cases. Using two levels has always lower request costs than using just one, and, when used with write combining, reduces the costs of all requests of an exchange below the worker costs in almost all configurations. Using three levels and write combining brings them to a negligible level in all configurations considered here.

Overall, the two optimizations give us effective knobs to reduce the costs due to requests to storage.

5.6 Extension to Broadcast

The techniques described in this section can be applied to the broadcast operator in a straight-forward manner, which is useful for broadcast joins. We leave the details as an exercise to the interested reader, but the main idea is to broadcast the input of each worker (without partitioning it) among the members of the same subgroups as defined for the exchange. For example, a two-level broadcast consists of a broadcast among the workers of the same column followed by a broadcast among those of the same row.

6 EVALUATION

6.1 Dataset and Methodology

In most experiments, we use the data sets of the TPC-H benchmark [42]. Since our prototype does not support strings yet, we modify `dbgen` to generate numbers instead of strings. At scale factor 1 k, the size of the data set is 502 GiB; in Parquet using standard encoding and GZIP compression, the size is 273 GiB. Following the best practices of big data processing and the systems we compare to below, we store the Parquet data in files of about 200 MB.

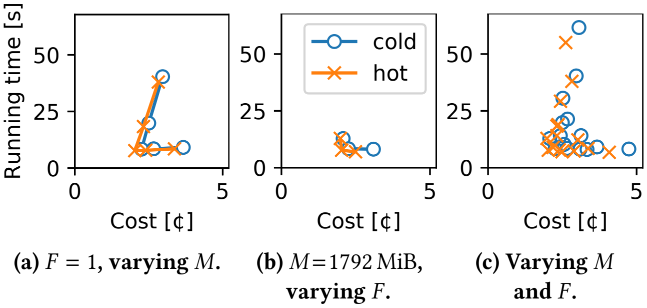
Unless otherwise mentioned, we measure the end-to-end query latency, which accounts for the serverless workers' invocation time, the useful work carried out, and fetching the results from the result queue in Amazon SQS. We report the median of three runs in the same data center, as we observed little variation across data centers in the experiments shown in Figures 4 and 5, as well as other isolated experiments not shown (with the exception of invocation into the cloud, as shown in Table 1).

We compare Lambada with two Query-as-a-Service systems, Google BigQuery [20] and Amazon Athena [19]. This type of system has a similar operational simplicity as Lambada, namely the ability to query data sets from cloud storage without starting or maintaining infrastructure, as well as a usage-based pricing model. QaaS is hence well suited for interactive analytics on cold data. In contrast, we do not consider Platform-as-a-Service solutions as they run on virtual machines that the user starts and stops and pays for while they are running. Examples include Amazon Redshift, Aurora, RDS, and its other managed DBMSs, Amazon EMR (Elastic MapReduce), and the corresponding offerings from the other providers.

6.2 Scan-heavy Queries

We first study the basic performance and cost characteristics on the two scan-heavy queries of TPC-H (*Q*1 and *Q*6) in order to avoid overlapping effects of more complex workloads. For the purpose of this section, we sort the `LINEITEM` relation by *l_shipdate* in order to show the effect of selection push-downs on that attribute.

6.2.1 Effect of Worker Configurations. In this experiment, we explore the parameter space of worker configurations to gain a deeper understanding of their impact. Specifically, we vary the amount of main memory of each worker, M, which influences the number of CPU cycles the function can use, as well as the number of files, F, that each worker processes. The latter parameter indirectly defines the number of workers: the tables is stored in 320 files, so we use $W = 320/F$ workers. We use TPC-H Query 1 (Q1), which selects 98 % of `LINEITEM` and aggregates them to a very small amount of groups, in order to eliminate effects of more complex plans. We create a fresh function for each configuration and each repetition and run the query twice, the first one as a *cold* run, the second as a *hot* run.



(a) $F = 1$, **varying** M. **(b)** M=1792 MiB, **varying** F. **(c) Varying** M **and** F.

Figure 8: TPC-H Query 1 with varying memory (M) and number of files (F) per worker.

Figure 8 shows the result. First, we fix the number of files per worker to $F = 1$ (i.e., $W = 320$) vary the memory size allocated per worker (512, 1024, 1796, 2048, and 3008 MiB). As Figure 8a shows, by increasing the worker size from 512 to 1796 MiB, execution gets significantly faster. This is because scanning GZIP-compressed data is CPU-bound and more memory means more CPU cycles as described in Section 4.1. Interestingly, it also gets marginally cheaper. We attribute this to the overhead of multi-threading in a configuration where that does not yield any gains and thus only reduces efficiency. As we increase the worker size further, the price increases (due to the linear relationship in the price model), however, without reducing running time. Similar to related work, we observe a small penalty on the end-to-end latency of cold runs of about 20 %. This is not only due to a slower invocation time, but also somewhat slower execution (possibly due to loading of code from the dependency layer), which affects the price. Despite of that, both hot and cold execution return in less than 10 s and thus within a timeframe that is suitable interactive analytics.

Figure 8b shows the results for varying the number of files per worker $F = \{4, 2, 1\}$, and with it the number of workers $W = \{80, 160, 320\}$, while fixing the worker size to $M =$ 1796 MiB. This is essentially the same experiment as the simulation from the Introduction shown in Figure 1a: using more workers speeds up execution, but at diminishing gains and thus increased costs. Finally, Figure 8c shows all different combinations of M and F. Which of the configurations (on the pareto-optimal front) a user might want to pick depends on her preference for price or performance and is out-of-scope of this paper. In the remainder of the paper, we either manually pick a good trade-off or show a range of configurations.

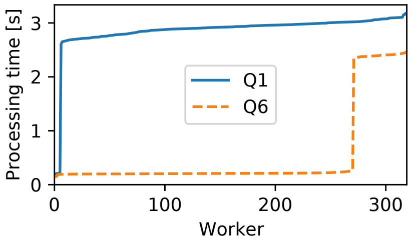


Figure 9: Distribution of processing time.

6.2.2 Effect of Push-downs. In order to study the effect of pushing down selections and projects into the scan operator, we run the two most scan-bound queries from TPC-H, Query 1 and 6. While Query 1 selects 98 % of the relation and uses seven attributes, Query 6 selects only 2 % of it but uses four attributes. In order to eliminate unrelated effects such as invocation time, we only measure processing time in this experiment, i.e., the time each worker takes to executes its plan fragment.

Figure 9 shows the processing time of all workers ordered by increasing processing time using $F = 1$ and $M = 1792$ MiB. In both queries, there are two categories of workers: those where the processing time is 100 ms to 200 ms and those where it is 2 s to 3 s. The workers of the former category load the metadata of their file (inducing one round-trip to S3), prune out all row groups due to the min/max indices on `l_shipdate`, and immediately return an empty result. For Query 1, this happens to about 2 % of the workers; for Query 6, to about 80 %, which corresponds to the respective selectivity of the filter on `l_shipdate`. If the min/max indices were stored in a central place and available before starting the workers, these workers would not even be started, but such optimizations are out of the scope of this paper. The other workers cannot prune out anything, so they load the projected columns from S3 and decompress and scan them. For them, the data volume of the projected columns determines the execution time, which is slightly higher in Query 1 than in Query 6.

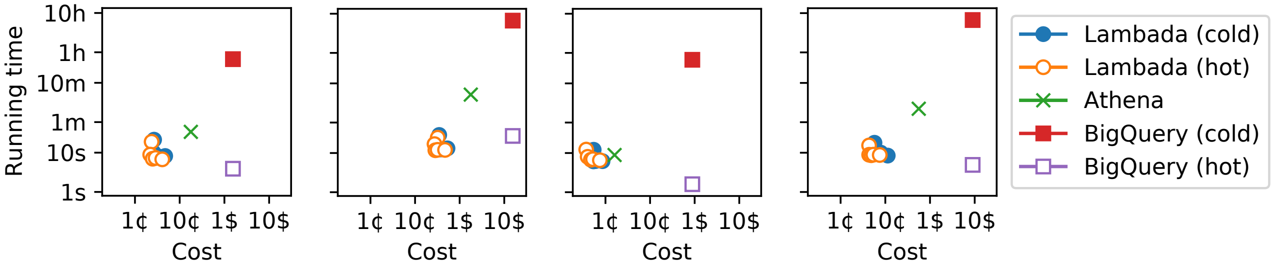
6.2.3 Comparison with QaaS Systems. We compare Lambada with two Query-as-a-Service systems, Google BigQuery [20] and Amazon Athena [19]. In practice, only Amazon Athena supports *in-situ* processing of large-scale datasets. Google BigQuery can currently only process individual files without prior loading (which is subject to further restrictions). Large-scale datasets need to be loaded with an ETL process, during which they are converted into a proprietary data format and possibly indexed. In this format, our `LINEITEM` table takes 823 GiB, which is slightly larger than the uncompressed CSV and over 5 × larger than our Parquet files. The promise of loading into this format is to allow for faster querying. We still include Google BigQuery in our study as the system otherwise fits well and the cloud provider could lift this restriction in the future. For Amazon Athena, we use the same files as for Lambada, which corresponds to the recommendations from the provider.

Both systems have a pay-per-query pricing model that is based on the number of bytes in the input relations and 1 TiB of input costs $5 in both systems. Only the bytes in attributes that are actually used in the query are taken into account and any type of computation including complex joins are free. However, selections are handled differently: in Google BigQuery all columns are always counted in their entirety, whereas in Amazon Athena only the selected rows of these columns are counted, i.e., selections are “pushed into the cost model.” Google BigQuery also charges per GB-month of loaded data, which we ignore in our comparison.

We run Lambada using one worker per file ($F = 1$), i.e., using 320 and 3200 workers for scale factors 1 k and 10 k, respectively. For Google BigQuery, we measure the time for loading the data, add that to the running time of the query and denote this time as “cold”; the query time alone is denoted “hot.” For Amazon Athena, we observed no noticeable difference between the first and subsequent runs, so we only show one number. The result is shown in Figure 10.

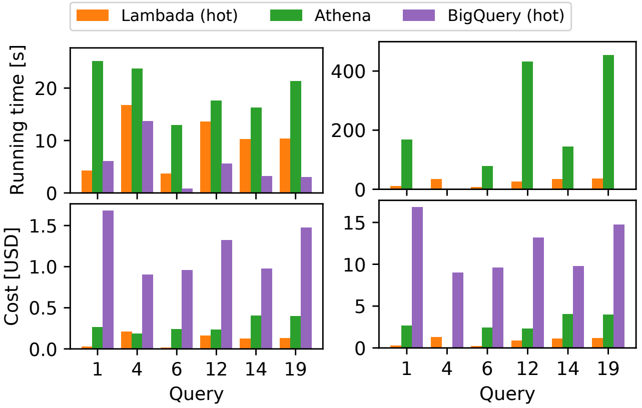
Running Time. In terms of end-to-end running time, Lambada is the system that has the most constant latencies. Since we use proportionally more workers as the data set grows, the pure processing time per worker stays constant and the latency only increases due to the (sublinearly) larger effort for invoking the workers, as well as a higher likelihood of stragglers and similar effects. In contrast, Amazon Athena does not seem to dedicate more resources for the larger data sets since their running time increases linearly. In BigQuery, the running time increases as well, though sublinearly, indicating that it uses somewhat more resources for the larger scale factor. We can only speculate why this is the case—at least for these simple queries, the cloud provider could also dedicate more machines for a shorter amount of time at an overall unchanged resource cost. In a system like Lambada, the user has more control and can thus increase the number of workers with the dataset size in order to get roughly constant query latencies.

In absolute terms, compared to Amazon Athena, the faster configurations of Lambada are about 4 × faster for Q1 and on par for Q6 at SF 1 k; at SF 10 k, Lambada is about 26 × and 15 × faster, respectively. Without taking data loading into account, Google BigQuery has running times as low as 3.9 s and 1.6 s for Q1 and Q6 at SF 1 k, respectively, and is thus significantly faster than Lambada. At SF 10 k, however, it is about 2.3 × slower and 2 × faster. Furthermore, the loading of the two scale factors takes about 40 min and 6.7 h, respectively. The loading does, hence, lead to faster querying, but at the price of a huge delay to the answer of the *first* query. Overall, the experiment shows that using serverless compute infrastructure is able to provide competitive performance compared to commercial Query-as-a-Service systems and is even able to outperform them, in some cases by large margins.



(a) Q1, SF 1 k. **(b) Q1, SF** 10 k. **(c) Q6, SF** 1 k. **(d) Q6, SF** 10 k.

Figure 10: Comparison of Lambada (using $F = 1$ and varying M) with commercial QaaS systems.



(a) SF 1 k, $W = 512$, $F = 2$. **(b) SF** 10 k, $W = 1280$, $F = 8$.

Figure 11: TPC-H queries on Lambada ($M = 2$ GiB).

Monetary Cost. For both queries and both scale factors, Lambada is cheaper than both other systems. Except for Q6 at SF 1 k, the difference is about one and two orders of magnitude compared to Amazon Athena and Google BigQuery, respectively. The difference to Google BigQuery is larger even though the price per TB is the same as that of Amazon Athena because the format of the former takes more space than that of the latter. In these cases, the serverless approach of Lambada is thus clearly more economic.

As expected, selections also have an influence on the cost. While the price of Q1 is essentially the same as that of Q6 in Google BigQuery (Q1 being slightly more expensive as it uses a few more attributes), the difference is significant in Amazon Athena. This is due to the different selectivities of the queries, which are taken into account in Amazon Athena's pricing model. In Q6, we only pay for the 2 % of the selected rows, while we pay for 98 % of them in Q1. For Q6, Lambada is thus only slightly cheaper than Amazon Athena. Lambada also benefits from the selectivity as discussed in the previous section, but not to the same degree. For queries with even more selective predicates, Amazon Athena would eventually become cheaper—up to the point where a query becomes free if it filters out all tuples in the input. Even for queries where the min/max filters of Parquet work perfectly, Lambada's cost could not be lowered below the cost of invoking other workers, loading the plan, fetching the metadata of each file, pruning out all row groups, and finally returning an empty result. In the most unfavorable case, highly selective queries that cannot benefit from min/max filters, Lambada would need to scan the entire input.

This discussion shows the role of the pricing model. While a serverless query processing system like Lambada runs on infrastructure that is rented *per unit of time* and has, thus, a monetary cost that is roughly proportional to the amount of resources used, the cost model of Query-as-a-Service systems is designed to be easily understandable by clients and, thus, extremely simple. It only needs to yield prices that are proportional to the resources used by the *overall workload mix* observed by the cloud provider. This means that some queries are necessarily under-priced while others are over-priced,[14] such as the scan-heavy queries in this section. For this type of query, a serverless solution like Lambada can have the biggest advantage.

6.3 End-to-End Workloads

6.3.1 TPC-H Queries. We now extend our evaluation to more complex queries. In addition to the two scan-heavy queries studied above, we implemented four other queries that use a variety of join variants and groupings. We compare against Athena and BigQuery as before. For BigQuery, we only run the queries at SF 1 k, as running them at a larger scale is out of our budget. However, we extrapolate the prices of SF 1 k to the larger scale factor, which should increase proportionally with the size of the input. For Lambada, we use a worker configuration that balances price and performance.

Figure 11 shows the result. The picture in this query mix is somewhat different than with the scan-heavy queries of the

[14] As we have shown in previous work [32], it is possible to exploit this pricing model by executing several queries at the price of a single one.

previous experiments. The advantage of Lambada over Athena at SF 1 k is now reduced: around 10 s for Lambada vs 15 s to 25 s for Athena and a similar difference for the prices. On the one hand, this is an achievement: even our research prototype is already competitive with a commercial product and further optimizations could improve both latency and price. On the other hand, it shows the limits of the usage-based pricing model of serverless functions when compared with the size-based pricing model of QaaS: While more complex queries run longer and, thus, increase the number of function-seconds the user is charged for, the price of QaaS systems is independent of the query complexity such that (almost) arbitrarily complex joins and groupings are essentially free. At SF 10 k, Athena does not seem to use more resources, so its running time is significantly higher, while Lamabada can keep the running time lower using more workers.

6.3.2 Scientific Workloads. As argued above, data analytics on serverless computing is most attractive for interactive workloads on cold data. This is a common pattern in the initial, exploratory phase of data analytics, when the user is getting to know the data set. To illustrate such cases, we apply Lambada to two scenarios from scientific domains, hydrology and high-energy physics (HEP), where this pattern is very common. In both domains, large amounts of massive data sets (often several terabytes) are shared through public repositories [4, 35]. These data sets are analyzed by research groups around the world, who use them in a variety of ways. The scientists running the analyses often work in small groups and query the data sets interactively in an ad-hoc fashion. These users thus need a system that can (a) scale to their data sets while remaining interactive and (b) is cost-effective for their infrequent usage.

Hydrology. We run the queries used in a study of Liu et al. [30] (described in more detail in [31]) to benchmark data analysis tools for that domain. The data used in hydrology usually consists of a number of measurements for each point in a grid with the three dimensions *longitude*, *latitude*, and *time*. The queries from the study consist of a spatial selection ($Q1$), a temporal selection ($Q2$), a regridding operation over a spatial selection ($Q3$), and computing historical values over a spatial selection ($Q4$ and $Q5$). The data set from the original study is not available, so we use one of the many publicly available ones that are similar in nature and structure: the multi-satellite precipitation product for the U.S. GPM team [18]. It contains ten different metrics related to rainfall recorded every 30 min since June 2000. We converted the data set to Parquet. After conversion, the data set consists of 2.77 TB.

We run the queries on Lambada using $W = 1500$ workers with $M = 2$ GiB and $F = 200$ files each, which we found to be a good trade-off in terms of responsiveness and cost. The results are shown in Figure 12. All queries run in less than 25 s and are thus interactive. Furthermore, they cost in the order of less than $1.3. The entire data set was not analyzed in the original study [30, 31], but projecting the fastest query runtime, Lambada improves at least by 5 ×. This makes the runtime go from 140 s to around 20 s.

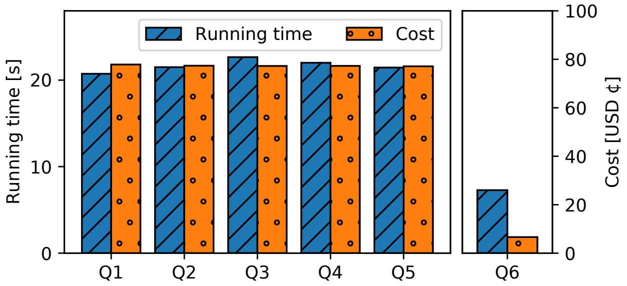


Figure 12: Hydrologist (Q1-Q5) and HEP queries (Q6).

High-Energy Physics. We run a query used by Cremonesi et al. [7] to study the applicability of big-data technologies for HEP tasks, which computes the distribution of the dimuon invariant mass. The data sets in this domain typically consist of massive collections of records called "events," each of which has several thousand fields, which are either scalar or nested inside both records and arrays. The query projects the input on a small number of nested fields, filters out events with less than two muons, selects the two most heavy muons of each of the remaining events, computes their invariant mass, and finally returns a histogram over that mass. We use the same data set [5] as the original authors (produced in the Large Hadron Collider), which contains all "runs" from 2010 RunB consisting of 2.6 TB of complex nested data.

We run this query on Lambada using $W = 1000$ workers with $M = 2$ GiB and $F = 29$ files each. The results are shown as $Q6$ in Figure 12. The query is run in less than 10 s and costs less than 10 ¢. Note that a much lower number of workers is sufficient compared to the other experiments. This is due to the fact that individual queries in this domain usually only select a tiny subset of the fields (a single-digit number out of several thousand). The ability of Lambada to download only the bytes belonging to the projected fields is therefore crucial as it allows it to skip over the majority of the input. In order to understand the improvement with [5], the dimuon invariant mass computation could be run in 438 s using 800 cores and 13 TB of main memory. Lambada performs this computation in under 10 s with $W = 1000$ workers, showing the efficiency of our system.

6.4 Exchange Operator

We compare the performance of our exchange operator in isolation with the numbers published for similar implementations

Table 3: Running time of S3-based exchange operators.

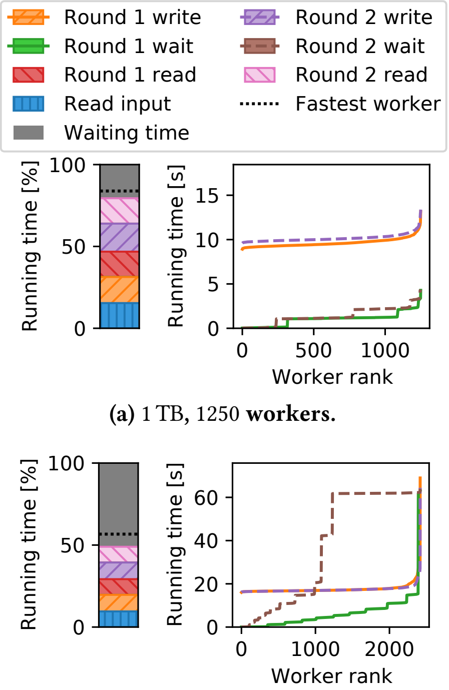
	#Workers	Storage Layer	
		VMs	S3
Pocket [27]	250	58 s	98 s
	500	28 s	
	1000	18 s	
Locus [38]	dynamic		80 s to 140 s
Qubole [41]	400		580 s
Lambada	250		22 s
	500		15 s
	1000		13 s

in previous work, namely Pocket [26], Qubol [41], and Locus [38]. We use a dataset of 100 GB because numbers are available for a dataset of that size for all other systems. Locus and Qubol use workers with 1536 MiB of main memory; Pocket uses 3008 MiB workers; for Lambada, we use 2048 MiB of allocated memory.

Table 3 shows the running time of the various approaches. Compared to the S3-based baseline implementation in the work on Pocket, Lambada runs 5 × faster on 250 workers. In contrast to that baseline, however, Lambada's sublinear amount of requests and the usage of multiple buckets enable it to scale to 500 and 1000 workers, which reduces running time further. Compared to the implementation using Pocket (i.e., using VM-based storage for intermediate results), Lambada is still 2.5 ×, 2 ×, and 1.4 × faster on 250, 500, and 1000 workers, respectively. Locus uses a dynamic number of workers and the paper does not detail the numbers for the experiment on 100 GiB, but even with 250 workers, Lambada is about 4 × faster than Locus' fastest configuration. Compared to both other systems, Lambada has the additional advantage of running without any always-on infrastructure. Qubole, essentially a serverless backend for Spark [44], is far off from all other systems, most likely due to the rate limits that the authors report.

In another experiment, we run the exchange operator on 1 TB and 3 TB datasets. It takes 56 s using 1250 workers for the former and 159 s using 2500 workers for the latter. On a dataset of 1 TB, Locus takes 39 s using a dynamic number of workers (which could be higher than what we use for Lambada), but uses VM-based fast storage for intermediate results.

For the larger dataset (3 TB and 2500 workers), waiting time for stragglers starts getting significant. Figure 13 gives details. The left sides of the plots show the fastest running time of each phase observed in any worker as a fraction of the end-to-end latency (which is dominated by the slowest worker). Plotting the fastest execution shows an informal lower bound for each phase. Note that reading the input, as well as writing the partition files and reading them again in each of the two phases, take exactly the same amount of time since they shuffle the same amount of data at full network bandwidth. Also note that the fastest waiting time is that of one round-trip to S3 (around 0.1 s), which is so short compared to the reading and writing that it is not visible in the plot. The dashed line shows the end-to-end running time of the fastest worker. On the 1 TB dataset, the fastest worker takes around 85 % of the slowest worker and is relatively close to the lower bound, i.e., to the sum of the fastest executions of the different phases. On the 3 TB dataset, the total execution time is more than 2 × as slow as it could be if all workers could run all phases at maximum speed; more than half of the total execution time is due to stragglers and waiting.



(a) 1 TB, 1250 **workers.**

(b) 3 TB, 2500 **workers.**

Figure 13: Break-down and per-phase running time distribution of TwoLevelExchange.

The right sides of the plots give details about the stragglers. For each phase, it gives a distribution of the running time of each worker ordered by increasing running time. We omit the three read phases as they do not experience significant tail latencies.[15] On both datasets, the write phases have a relatively stable running time until the 95-percentile. The slowest worker,

[15] This is not the case when using the default configuration. Instead, aggressive timeouts and retries are necessary to reduce tail latencies, but describing such optimizations in detail are out of scope of this paper. Then

however, is about 30 % and 4 × slower than the median for the small and big datasets, respectively. These latencies propagate: The waiting time in the first round is significant for a large number of workers because each worker that is slow with writing causes wait time for all workers in its group. In turn, those workers start later with the next phase and thus cause wait time for even more workers. While the wait time is moderate for the small dataset, it dominates the execution time of the larger one. Further research is required to reduce the tail latencies appearing at these scales. Nonetheless, our experiments show that exchange operators can be implemented under a purely serverless paradigm and even outperform approaches with always-on infrastructure.

7 RELATED WORK

Our work has two main lines of related work: data analytics on cold data and serverless computing.

Data analytics on cold data. Performing data analytics over cold data has been studied extensively by both academia [8, 10, 28] and industry [22, 23]. Topics range from techniques for avoiding to fetch data from cold storage [1] to using cold storage devices as cheap storage for intermediate results [3]. However, all of this work is done in the context of long-running systems that are typically maintained by a dedicated department of a large institution and therefor targets a somewhat different use case than serverless computing.

Serverless computing in general. Recently, there have been many systems design proposals that leverage serverless functions in different settings. Examples include distributed make [11, 27], sorting [24, 27, 38, 40], video encoding [2, 11, 12], image and video classification [2, 11, 27], unit tests [11], as well as MapReduce-style [24, 25, 38, 40] and SQL-style analytics [38]. We relate to these works in the sense they also use serverless workers for cost-efficiency at infrequent usage.

Fouladi et al. [11] also observe the invocation bottleneck. The present several techniques to increase the invocation rate from what we call the driver, which reduces the startup time of 1000 workers to around 6 s. As discussed in Section 3, this may be too long for even larger numbers of workers. The two-level mechanism we propose instead manages to start several thousand workers in under 4 s.

Serverless data analytics systems. The most similar work to ours may be Flint [25]. The authors propose a rewrite of the Apache Spark execution layer using serverless workers and cloud storage. However, its query execution time is more than 10 × higher for similar queries run in Lambada. For instance, Flint takes around 100 s for scanning a 1 TB of data whereas Lambada would take 10 s with the same amount of workers. This shows the difference between designing a system from scratch and refitting an existing one and that the latter may introduce inefficiencies that overcompensate the potential economic advantage of serverless. Even more similar is the work on Starling [37], which was concurrent to ours and presented at the same venue.

It is also possible to use general-purpose serverless frameworks such as PyWren [24] for data analytics. However, the user would then have to re-implement most of the techniques we describe in this paper as part of the query logic, which is much less convenient and likely to be less efficient.

Data exchange in serverless systems. Previous work has proposed solutions for data exchange in the serverless context [26, 27, 38]. For example, Klimovic et al. [27] design an elastic, fast, and fully managed storage system for ephemeral data. Similarly, Pu et al. [38] design a system for intermediate data that uses a combination of AWS ElasticCache and AWS S3. Their motivation, however, lies in the limitations of the basic operator and its quadratic number of requests. As we show, this is not a fundamental problem of exchange. Furthermore, the solutions consist of additional services, which compromises the advantages of a serverless system.

Critique of serverless computing. Hellerstein et al. [17] argue that serverless functions are not suitable for data analytics. By building a full-fledged query processing system we show that the serverless paradigm is, in fact, viable for interactive analytics on cold data.

At the same time, we agree that serverless *functions* do have short-comings for job-oriented applications like Lambada, as well as all other work on serverless mentioned in this section. We argue that, instead, this type of application would require "serverless *clusters*," a concept we proposed in [33].

8 CONCLUSIONS

In this work, we show that data analytics on serverless computing is technically possible and economically viable for interactive use on cold data. Through the implementation of a full-fledged system, Lambada, we identify a number of challenges and propose solutions for them: tree-based invocation of workers for fast start-up, a design for scan operators that balances cost and performance of cloud storage, and a purely serverless exchange operator. The latter overcomes limitations that were previously thought to be inherent to the serverless paradigm. Thanks to our optimizations, Lambada can answer queries on more than 1 TB of data in about 15 s, which makes it competitive with commercial Query-as-a-Service systems and an order of magnitude faster than job-scoped VM infrastructure.

basic idea is described by Amazon's "Performance Guidelines for Amazon S3" at https://docs.aws.amazon.com/AmazonS3/latest/dev/optimizing-performance-guidelines.html#optimizing-performance-guidelines-retry.

REFERENCES

[1] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. “Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia.” In: *PVLDB* 6.14 (2013). doi: 10.14778/2556549.2556556.

[2] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. “Sprocket: A Serverless Video Processing Framework.” In: *SoCC*. 2018. doi: 10.1145/3267809.3267815.

[3] Renata Borovica-Gajić, Raja Appuswamy, and Anastasia Ailamaki. “Cheap Data Analytics using Cold Storage Devices.” In: *PVLDB* 9.12 (2016). doi: 10.14778/2994509.2994521.

[4] CERN. *CERN Open Data Portal*. url: http://opendata.cern.ch/ (visited on 01/20/2020).

[5] CERN. *MuOnia primary dataset in AOD format from RunB of 2010*. doi: 10.7483/OPENDATA.CMS.TME9.7FP2. (Visited on 01/20/2020).

[6] Microsoft Corp. *Azure Functions*. url: https://azure.microsoft.com/en-us/services/functions/ (visited on 10/19/2019).

[7] Matteo Cremonesi et al. “Using Big Data Technologies for HEP Analysis.” In: *CHEP*. 2019.

[8] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. “Anti-Caching: A New Approach to Database Management System Architecture.” In: *PVLDB* 6.14 (2013). doi: 10.14778/2556549.2556575.

[9] David DeWitt and Jim Gray. “Parallel Database Systems: The Future of High Performance Database Systems.” In: *CACM* 35.6 (1992). doi: 10.1145/129888.129894.

[10] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. “Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database.” In: *PVLDB* 7.11 (2014). doi: 10.14778/2732967.2732968.

[11] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. “From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers.” In: *USENIX ATC*. 2019.

[12] Sadjad Fouladi et al. “Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads.” In: *NSDI*. 2017.

[13] G. Graefe and D.L. Davison. “Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution.” In: *IEEE Trans. Softw. Eng.* 19.8 (1993). doi: 10.1109/32.238579.

[14] Goetz Graefe. “Encapsulation of Parallelism in the Volcano Query Processing System.” In: *SIGMOD*. 1990. doi: 10.1145/93597.98720.

[15] Goetz Graefe. “Query Evaluation Techniques for Large Databases.” In: *CSUR* 25.2 (1993). doi: 10.1145/152610.152611.

[16] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing*. 2nd Edition. Addison-Wesley, 2003. isbn: 9780201648652.

[17] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. “Serverless Computing: One Step Forward, Two Steps Back.” In: *CIDR*. 2019.

[18] G.J. Huffman, E.F. Stocker, D.T. Bolvin, E.J. Nelkin, and Jackson Tan. *GPM IMERG Early Precipitation L3 Half Hourly 0.1 degree x 0.1 degree V06*. Goddard Earth Sciences Data and Information Services Center (GES DISC). doi: 10.5067/GPM/IMERG/3B-HH-E/06. (Visited on 01/20/2020).

[19] Amazon Inc. *Amazon Athena*. url: http://docs.aws.amazon.com/athena/ (visited on 10/19/2019).

[20] Google Inc. *Google BigQuery*. url: https://cloud.google.com/bigquery/ (visited on 10/19/2019).

[21] Google Inc. *Google Cloud Functions*. url: https://cloud.google.com/functions/ (visited on 10/19/2019).

[22] IBM Inc. *IBM Multi-temperature management*. url: https://www.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.admin.dbobj.doc/doc/c0059106.html (visited on 10/19/2019).

[23] SAP Inc. *SAP using Spark to process cold data next to a main memory database*. url: https://blogs.saphana.com/2018/12/03/what-is-sap-hana-cold-data-tiering/ (visited on 10/19/2019).

[24] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. “Occupy the Cloud: Distributed Computing for the 99%.” In: *SoCC*. 2017. doi: 10.1145/3127479.3128601.

[25] Youngbin Kim and Jimmy Lin. “Serverless Data Analytics with Flint.” In: *CLOUD*. 2018. doi: 10.1109/CLOUD.2018.00063.

[26] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. “Understanding Ephemeral Storage for Serverless Analytics.” In: *NSDI*. 2018.

[27] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. “Pocket: Elastic Ephemeral Storage for Serverless Analytics.” In: *OSDI*. 2018.

[28] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. “Identifying Hot and Cold Data in Main-Memory Databases.” In: *ICDE*. 2013. doi: 10.1109/ICDE.2013.6544811.

[29] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy Lohman. “NUMA-aware algorithms: the case of data shuffling.” In: *CIDR*. 2013.

[30] Haicheng Liu, Peter Oosterom, Chengfang Hu, and Wen Wang. “Managing Large Multidimensional Array Hydrologic Datasets: A Case Study Comparing NetCDF and SciDB.” In: *Procedia Engineering* 154 (2016). doi: 10.1016/j.proeng.2016.07.449.

[31] Haicheng Lui. “Comparing NetCDF and a multidimensional array database on managing and querying large hydrologic datasets: A case study of SciDB.” MA thesis. TU Delf. (Visited on 10/19/2019).

[32] Renato Marroquín, Ingo Müller, Darko Makreshanski, and Gustavo Alonso. “Pay One, Get Hundreds for Free: Reducing Cloud Costs through Shared Query Execution.” In: *SoCC ’18*. doi: 10.1145/3267809.3267822.

[33] Ingo Müller, Rodrigo Bruno, Ana Klimovic, John Wilkes, Eric Sedlar, and Gustavo Alonso. “Serverless Clusters: The Missing Piece for Interactive Batch Applications?” In: *SPMA*. 2020. doi: 10.3929/ethz-b-000405616.

[34] Ingo Müller, Renato Marroquín, Dimitrios Koutsoukos, Mike Wawrzoniak, Sabir Akhadov, and Gustavo Alonso. *The Collection Virtual Machine: An Abstraction for Multi-Frontend Multi-Backend Data Analysis*. 2020. arXiv: 2004.01908 [cs.DB].

[35] NASA. *DATA.NASA.GOV: A catalog of publicly available NASA datasets*. url: http://data.nasa.gov/ (visited on 01/20/2020).

[36] M. Tamer Özsu and P Valduriez. *Principles of Distributed Database Systems*. 3rd ed. Springer, 2011. ISBN: 9781441988331.

[37] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. "Starling: A Scalable Query Engine on Cloud Function Services." In: *SIGMOD*. 2020.

[38] Qifan Pu, U C Berkeley, Shivaram Venkataraman, Ion Stoica, U C Berkeley, and Implementation Nsdi. "Shuffling , Fast and Slow: Scalable Analytics on Serverless Infrastructure." In: *NSDI*. 2019.

[39] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. "High-Speed Query Processing over High-Speed Networks." In: *PVLDB* 9.4 (2015). DOI: 10.14778/2856318.2856319.

[40] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. "Serverless data analytics in the IBM cloud." In: *Middleware Industry*. 2018. DOI: 10.1145/3284028.3284029.

[41] Venkat Sowrirajan, Bharath Bhushan, and Mayank Ahuja. *Qubole offers Apache Spark on AWS Lambda*. 2017. URL: https://www.qubole.com/blog/spark-on-aws-lambda/ (visited on 12/20/2019).

[42] Transaction Processing Performance Council. *TPC Benchmark H (Revision 2.18)*. 2018.

[43] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. "Peeking Behind the Curtains of Serverless Platforms." In: *USENIX ATC*. 2018.

[44] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing." In: *NSDI*. 2012.