A Serverless Implementation of MapReduce on AWS Lambda

Saarthak Gupta, Agi Luong
University of Virginia
uzn2up, xwq5ja@virginia.edu

Abstract

MapReduce is a scalable computation model that allows batch processing of enormous datasets. Serverless computing is a cloud programming paradigm in which software can be deployed with resources allocated on-demand without the need to manage server infrastructure. We wanted to bring these two models together and create a Serverless MapReduce implementation that provides the advantages of both worlds, like cost-effectiveness, elasticity, and parallelism. We used AWS Lambda to invoke cloud functions that perform map or reduce tasks and AWS S3 as a distributed object store for inputs, outputs, and intermediate data. Our implementation is able to scale to thousands of concurrently executing Lambdas. We tested out the classic word count MapReduce example for a wide range of input data set sizes. In our testing, our Serverless runtime is able to take advantage of the highly scalable nature of AWS Lambda. The smallest test case ran in under 10 seconds, and the largest, with about 1,000 concurrently executing functions, in about 800 seconds. We conclude that Serverless environments are feasible and effective for running MapReduce-style jobs, and further work in this area can lead to exceptional production systems.

1 Introduction

MapReduce is a massively parallel, distributed computation paradigm pioneered by Google [2] in response to their growing need for batch processing of data. The user of the model defines two functions: Map: (Key, Value) $\rightarrow$ (Key, Value) and Reduce: (Key, value list[]) $\rightarrow$ output. The map phase performs filtering and sorting, and the reduce phase performs a summary operation to yield the final output.

Historically, hosting applications on the internet requires provisioning and managing physical or virtual servers. More recently, an advent has been observed in Function-as-a-service (FaaS) applications, which is a serverless architecture that allows the execution of code in response to events without having to build out any complex server infrastructure. We want to adapt MapReduce to this new computation architecture.

Traditionally, MapReduce is performed on a multi-node cluster that requires huge investment for hardware and supporting infrastructure in data centers like networking, power, cooling, etc. (building data centers can cost $125-$200+ per sq. foot). Even the costs of renting out servers from a company like Amazon can add up quickly – a one-year EC2 Instance Savings Plan works out to $1,060. On the other hand, AWS Lambda is very cost-effective at only $0.20 per one million requests, and S3 buckets cost $0.023 per GB. Combined with the dynamic scalability of FaaS, a serverless service like AWS Lambda seems like a perfect platform to implement MapReduce.

In this paper, we describe our implementation of MapReduce with this Serverless design paradigm in mind. In section 2, we focus on the architecture of our implementation and how various components work together to make MapReduce work. In section 3, we present our testing framework and performance evaluation of our implementation. In section 4, we discuss other related work and methodologies. Finally, in section 5 we draw conclusions based on what we learned while implementing MapReduce Severlessly and section 6 has links to our open-source implementation.

2 Architecture

In this section, we describe the architecture of our serverless MapReduce runtime and its functioning. The entire application consists of various parts like the client-side code, AWS S3 buckets, AWS Lambda functions and their associated code, and ancillary scripts for evaluation and testing. All code is written in Python. A detailed overview of the architecture is shown in Figure 1, with six mappers and two reducers.

2.1 Components

In order to make a Serverful MapReduce implementation work, two main components are needed: machines (or "nodes") that run the map or reduce jobs and an underlying

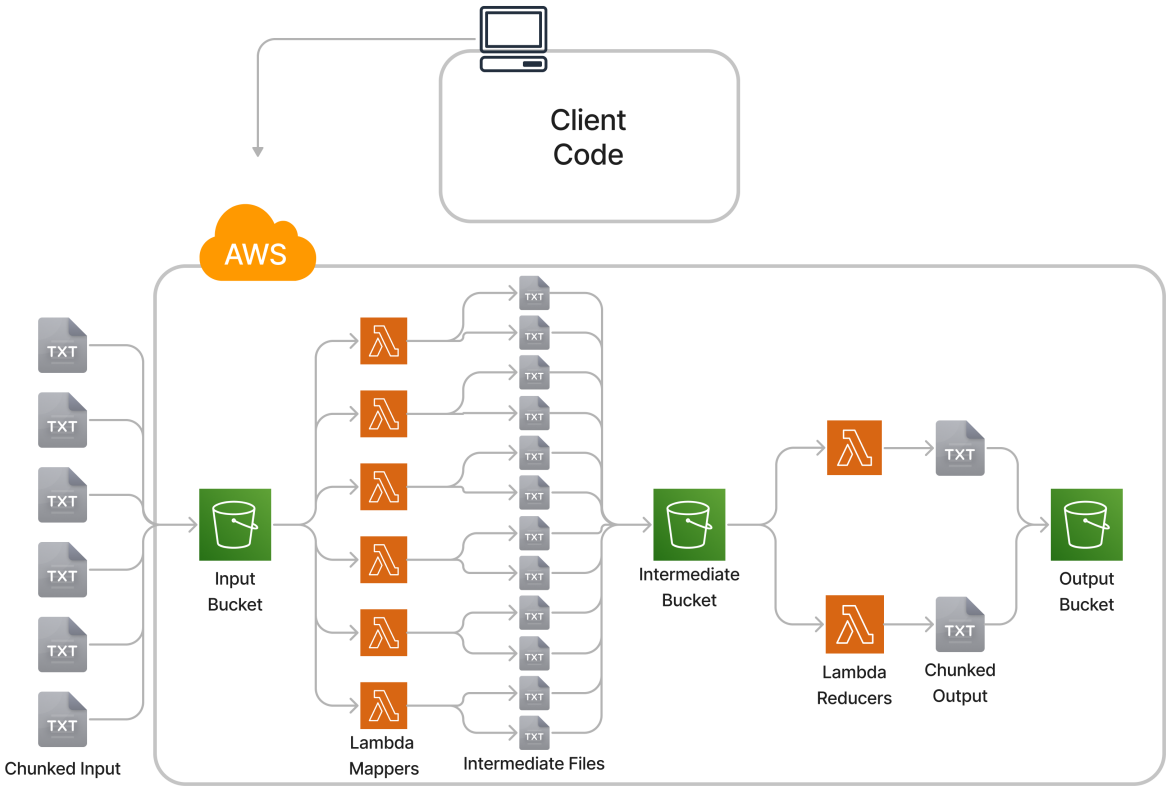

Figure 1. Architecture of the Serverless MapReduce runtime

distributed file system for sharing data. Google's implementation of MapReduce used the Google File System [3] as the distributed file system. In our implementation, we used AWS S3- Amazon's highly scalable object store- as our distributed file system. Instead of long-running servers that run map or reduce tasks, our Serverless implementation will use short-lived AWS Lambda functions. Just like any other MapReduce implementation, the user defines *Map(Key, Value)* and *Reduce(Key, Value List)* functions based on their specific task.

The Serverless MapReduce implementation is supported by the following main components:

- *Input Bucket*: The input bucket stores the input corpus chunked into many files. The number of mappers invoked (m) is equal to the number of chunks the input is divided into. Since Lambda is built for concurrent executions, this allows us to scale our workloads in a massively parallel manner by chunking the input as much as possible, beyond what may be possible for a Serverful implementation.

- *Lambda Mappers*: When a Lambda Mapper is invoked, it is passed the input file and the number of reducers (n) defined by the client. The mapper calls the user-defined *Map(Key, Value)* on the file contents to obtain the intermediate key-value pairs. The mapper bins the intermediate key-value pairs based on the equation:

$$bucket = hash(key) \pmod{n}$$

Each of the n intermediate files is encoded as JSON and stored in the intermediate bucket.

- *Intermediate Bucket*: Each Mapper Lambda emits n intermediate files that contain JSON-encoded key-value pairs and are stored in the intermediate S3 bucket. The total number of intermediate files for a given MapReduce job is $m \times n$. The files follow the naming convention:

$$ReducerID\_InputChunkUID.json$$

Here, $ReducerID \in [0, n-1]$ and $InputChunkUID$ is a unique identifier associated with each input chunk. This naming convention is used by the runtime to figure out

what reducers can be run and the files that need to be assigned to each reducer.

- *Lambda Reducers*: When a Reducer Lambda is invoked, it is passed a list of all the intermediate files it needs to read and the *ReducerID*. After reading all the relevant JSON data, the reducer transforms the key-value pairs into (key, value list) format. That is, for each unique key, we have a list of all the values associated with that key. The user-defined *Reduce(Key, Value list)* function is called for each unique key to obtain the final output, which is encoded as JSON.

- *Output Bucket*: Each of the n reducers writes one file to the output bucket. To obtain the aggregate output of the MapReduce job, all the output chunks may have to be combined. If the output is in key-value pair form, it may be used as the input for another MapReduce job.

- *Serverless MapReduce Client*: The Serverless MapReduce Client is the application the user interacts with to submit their MapReduce job. The main parameters the user has to specify are the S3 input bucket that contains the chunked input files and the number of reducers desired (n).

- *Miscellaneous Components*: Other parts of the system include AWS IAM roles and permission policies that allow communication between the various AWS services and ancillary components for combining output bucket data and comparing it against known values to validate the system.

Each Lambda was allocated a memory of 1,024 MB, 512 MB storage, and a timeout of 10 minutes.

2.2 Workflow

A typical run of a Serverless MapReduce job looks as follows:

1. The user specifies their input bucket and the number of reducers (n) and runs the MapReduce client.

2. The MapReduce client invokes a Mapper Lambda for each input chunk. This can potentially be thousands of mappers.

3. The Mapper Lambda stores its intermediate files (containing intermediate key-value pairs) in the intermediate bucket.

4. Depending on what intermediate files have been written to the intermediate bucket, the MapReduce client figures out if any Reducer Lambdas are ready to run and invokes them until all n reducers have been invoked.

5. Once all the reducers are done, the chunked output is stored in the output S3 bucket. At this point, the output may be aggregated or passed on to another MapReduce job.

3 Testing

To evaluate the correctness of our MapReduce implementation and benchmark its performance, we decided to use the classic word count MapReduce task. We used books from the Project Gutenberg library to create input data sets of various sizes for our testing. The user-defined map and reduce functions for this task are presented below. This is the only thing the user has to specify in addition to the input files and number of reducers.

Algorithm 1 Map Function For Word Count

```
1: function UserMapFunction(k, v)
2:     words ← [filtered words from v]
3:     kv_list ← []
4:     for word in words do
5:         kv_list.append((word.lower(), 1))
6:     end for
7:     return kv_list
8: end function
```

Algorithm 2 Reduce Function For Word Count

```
   function UserReduceFunction(k, val_list[])
2:     return len(val_list)
   end function
```

3.1 Methodology

Using different books from the Project Gutenberg repository, we created test cases of increasing sizes. The number of Mapper Lambdas invoked is equal to the number of input files or chunks, and the number of Reducer Lambdas invoked is specified by the user. We created three total test cases: small, medium, and large. These scale from tens of Lambdas to about a thousand concurrently executing Lambda functions.

The different testing scenarios and their average completion times are presented in Table I. The current configurations allow 1,000 concurrent Lambda invocations, so some throttling was observed in the largest test case.

Each test scenario was run five times to obtain the average completion time across all runs. The number of reducers used was scaled appropriately based on how many input chunks each test scenario had. Making the number of reducers too small would result in Lambda invocations timing out as each reducer was overloaded with a lot more work, causing it to use up all its working memory and not finish before the timeout.

Table I. Average Time to Complete MapReduce Jobs for Different Input Datasets						
Test	Number of Files	Average File Size (KB)	Total Input Data Size (MB)	Number of Mappers	Number of Reducers	Average Time To Completion (s)
Small	8	412	3.3	8	6	6.12
Medium	98	628	61.6	98	80	26.44
Large	932	687	644.1	932	800	864.77

3.2 Results

The results in Table I demonstrate the highly scalable nature of implementing MapReduce on AWS Lambda and S3. The average job completion times we observed were small and scaled well as the size of the job increased. The smallest job took under 10 seconds to complete. The largest test case invokes almost 1,000 concurrently executing lambdas and writes 745,600 files to the intermediate bucket and 800 files to the output bucket. The total number of lambda invocations for the large test case is 1,732. The largest test case demonstrates the performance of our MapReduce runtime at scale. This test case was able to produce the result of the word count job in less than 15 minutes.

The main reason for the low average completion time, even in the largest case, is due to the massive parallelism that is allowed by concurrently executing Lambda functions. Also, S3 buckets provide rapid IO capabilities. In Section 5, we discuss methods to scale this even more in the future by increasing Lambda concurrency limits, and sub-dividing S3 buckets.

4 Related Work

A similar project called MARLA [4] to implement Serverless MapReduce was undertaken by the GRyCAP research group. The biggest difference between their architecture and ours is that they use a coordinator Lambda function that is triggered on an S3 bucket upload event. This coordinator Lambda is responsible for spawning Mappers, checking what Reducers are ready to run, the shuffle stage, and running the Reducers. In this architecture, the coordinator Lambda is likely to serve as a bottleneck for large jobs, as Lambda functions have a maximum runtime of 900 seconds and a maximum memory of 10,240 MB.

Another project called Corral [1] implements a similar architecture to what we have. It offers two ways to run the MapReduce job, either with a Coordinator Lambda function or using client-side code running locally on the user's machine. This implementation is written in Go and takes advantage of thread-level parallelism, which is not possible in a Python implementation. Due to this, it may offer faster performance overall for the coordinator's implementation.

5 Conclusion

The implementation of a Serverless MapReduce framework on AWS Lambda presents a promising approach to handling large-scale data processing tasks efficiently and cost-effectively. Through our testing and evaluation, we have observed several key findings that support the feasibility and effectiveness of this approach for production workloads.

Firstly, the performance of our Serverless MapReduce implementation has demonstrated scalability and rapid execution times across varying sizes of input datasets. Even under considerable loads, with nearly 1,000 concurrently executing Lambdas, our framework maintained relatively low average completion times. This highlights the inherent advantage of leveraging the highly parallel nature of AWS Lambda functions and the rapid IO capabilities of AWS S3.

5.1 Cost

The cost analysis reveals the economic benefits of utilizing a Serverless architecture for MapReduce tasks. By leveraging AWS Lambda and S3, the operational costs associated with hardware provisioning and maintenance are significantly reduced. The pay-per-invocation model of AWS Lambda and the low storage costs of S3 contribute to a cost-effective solution for batch processing of large datasets.

Based on current Lambda and S3 prices, the cost of a single job can be calculated as:

$$\$0.0000002 * invocations + \$0.00001667 * GB\_seconds$$

$$+ S3\_cost$$

Data transfer between AWS Lambda and S3 is free in the same region, which makes the S3 cost very small. The cost of the Lambdas depends on both the number of invocations and the duration the function runs for (GB-seconds). Based on the number of mapper and reducer Lambda invocations as well as the average running duration, we calculated the AWS Lambda cost associated with each test case. Then, based on the amount of data stored in the input, intermediate, and output buckets, we calculated the S3 cost. Finally, we added these for each test case to obtain an estimated cost. Table II shows the estimated cost for each test case. See Appendix A for the data that was used to estimate these costs.

Table II. Average Estimated Cost Of Job	
Test	Estimated Cost
Small	$0.0003
Medium	$0.104
Large	$0.514

5.2 Other Design Considerations

Our evaluation also identifies potential areas for optimization and future enhancements. By exploring strategies such as increasing Lambda concurrency limits and subdividing S3 buckets, we can further enhance the scalability and performance of our Serverless MapReduce framework. Additionally, considerations such as managing Lambda cold starts and handling failures can be addressed to ensure robustness and reliability in production environments. There are many design considerations to be made for a production system. We offer an overview of such conditions in this section.

AWS S3 buckets allow very high throughput; however, for a large enough data set, bucket IO may become a bottleneck. Our proposal to overcome this is to further divide the input, intermediate, and output buckets into multiple sub-buckets. In our current design, we use a single S3 bucket for each level of IO. Lambda functions can have a cold-start cost associated with them, as AWS may unload the function between invocations. A naive way to overcome this is to periodically schedule the Lambdas to execute even if there is no real workload in order to keep them primed. AWS also provides the option of provisioned concurrency, which allows pre-initialized execution environments for Lambda functions.

Another consideration is Lambda failures. These are rare but may occur due to a function hitting its memory or time limits. To deal with these, the runtime can be modified to monitor for these errors by looking at the files output to the S3 buckets. If a file does not appear within a timeout, the map or reduce task may be re-scheduled. Stragglers can be dealt with in a similar way by treating slow Lambdas as failed Lambdas when the job is nearing completion. The timeout for these should be lower than the maximum timeout for the Lambda function. AWS poses some limits on how many concurrent Lambdas can be executed, which may act as a bottleneck to scalability. However, these limits can be increased by special requests.

6 Metadata

The code/data of the project can be found at:

https://github.com/saarthak2002/serverless-mr

The link to our video presentation:

https://github.com/saarthak2002/serverless-mr

References

[1] Ben Congdon. Introducing Corral: A Serverless MapReduce Framework. https://benjamincongdon.me/blog, May 2, 2018.

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Dec 2004.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Oct 2003.

[4] GRyCAP Research Group. MARLA - MApReduce on AWS Lambda. https://github.com/grycap/marla, Accessed: 2024. GRyCAP research group of the Institute for Molecular Imaging Technologies at the Universitat Politècnica de València, Spain.

Appendix A

Average S3 Storage Used And Lambda Runtime For Different Test Cases							
Test	Input bucket data size	Intermediate bucket data size	Output bucket data size	Number of mapper lambda invocations	Number of reducer lambda invocations	Avg runtime of mapper lambda	Avg runtime of reducer lambda
Small	3.3 MB	7.7 MB	528.1 KB	8	6	26500	20000
Medium	61.6 MB	120.2 MB	7.4 MB	98	80	37000	27000
Large	644.1 MB	1100 MB	156.7 MB	932	800	37000	27000