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 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.

[**TODO**: Abstract will be modified later with testing methodologies, results, and conclusions; our main goal for the checkpoint was to get our MapReduce implementation up and running.]

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 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

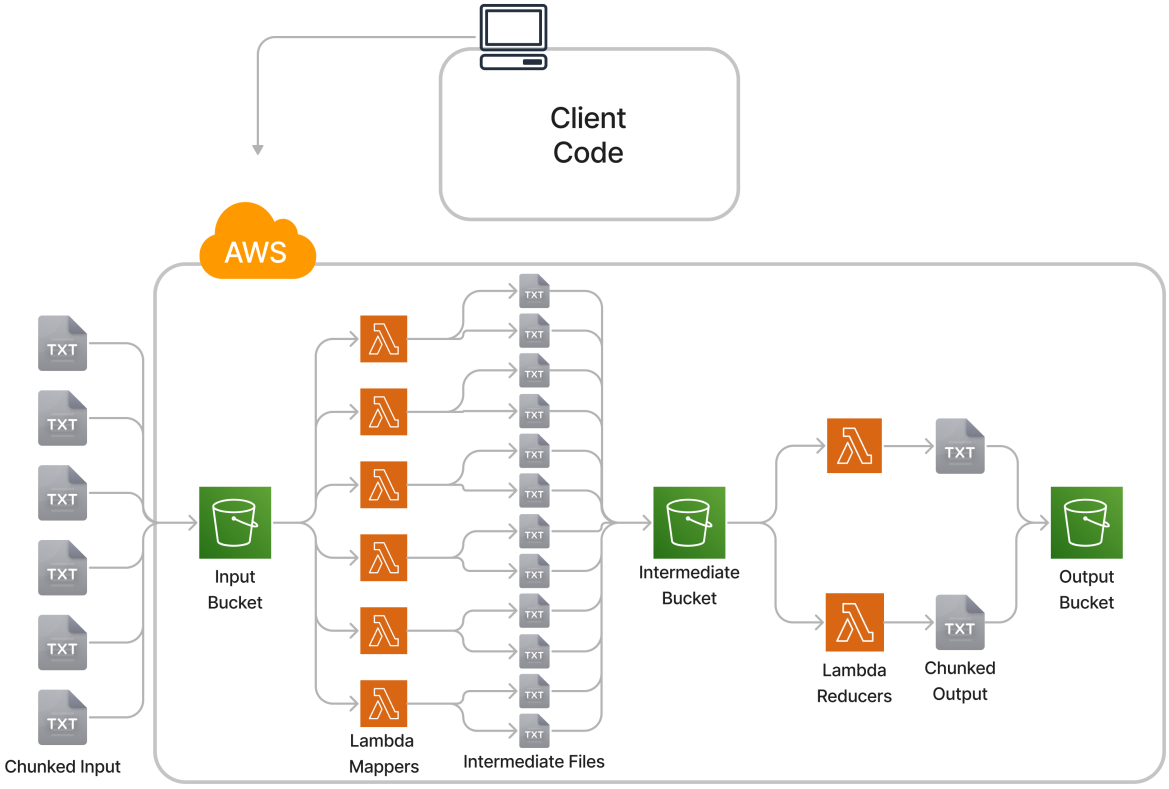

Figure 1. Architecture of the Serverless MapReduce runtime

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 256 MB, 512 MB storage, and a timeout of 2 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

[**TODO**: We have tested the validity of our implementation and it produces correct results for the project Gutenberg dataset. We plan to benchmark the performance of our implementation and compare it to other MapReduce implementations.]

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

[**TODO**: Draw conclusions about the feasibility of Serverless MapReduce based on the implementation process and the results obtained during the benchmarking.]

6 Metadata

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

`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.