Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Serverless Data Pipelines

Author:
Hang Li Li

Supervisor:
Dr. Robert Chatley
Second Marker:
Dr. Giuliano Casale

June 15, 2020

Submitted in partial fulfilment of the requirements for the MEng Computing of Imperial College London

Abstract

We present ServerlessMR, a Big Data MapReduce framework that leverages AWS Lambda to allow processing of large data sets in a fully serverless architecture. It provides support for multi-stage and multi-pipeline tasks such as map-reduce-map-reduce-map, and allows for local and completely serverless executions of a task. The high elasticity of serverless computing allows ServerlessMR to efficiently support jobs whose resource requirements vary significantly during their execution.

ServerlessMR solves the problems that come with server-oriented Big Data frameworks such as Hadoop and Spark, which require complex cluster setup, management and configuration. Long-running clusters in such frameworks are often under-utilised, leading to users incurring excessive costs as charges accumulate even when their clusters are idle.

ServerlessMR has been evaluated against Hadoop using the Berkeley Big Data Benchmark. Although Lambda is more expensive than a long running VM on a second-by-second basis, our experimental results suggest that using ServerlessMR will be cheaper than a long-running Hadoop cluster for ad-hoc analytics, short-running tasks and any usage where the cluster utilisation is below 30%. What is more, we believe that this threshold is also broadly applicable to other serverless Big Data frameworks.

Acknowledgements

I would like to thank my supervisor Dr. Robert Chatley for the incredible amount of guidance and support that he has provided me throughout the project. The key to the success of this project has been due to the ingenious ideas that he has provided and the discussions with him have been the highlight of my final year at Imperial.

Furthermore, I would like to thank my family for always being there for me in difficult moments over these four years. Without them, I wouldn't have achieved all this.

And, last but not least, I would like to thank my friends, who I share lots of lasting memories with, throughout these four years at Imperial.

Contents

Chapter 1

Introduction

1.1 Motivation

In the era of Big Data, frameworks such as Apache Hadoop and Apache Spark have become increasingly popular for large-scale data processing. Executing data processing jobs using these frameworks requires dedicated clusters of machines. Nodes in a cluster tend to be virtual machines from either cloud computing services such as AWS[1] or an organisation's own data centres. Using dedicated clusters of machines for Big Data processing is computationally fast, however, it presents two major drawbacks:

- Many users are left to struggle with complex cluster setup, management and configuration, even for running simple embarrassingly parallel jobs. For instance, for every job, users need to determine a suitable number of machines for the job's cluster. Over-provisioning will result in having idle nodes during the job execution and incur a higher cost than necessary. Under-provisioning will normally slow down the execution considerably, but sometimes can even cause it to fail. The appropriate amount of resources for a job can only be obtained through a trial and error approach. Moreover, this means that a cluster that is well-tuned for one job might not be suitable for others. Furthermore, the dataset of a job can also affect its required cluster size.

- Often, a cluster has idle time and thus, a higher than necessary cost is incurred due to the inefficient use of the provisioned computing resources. This is common in ad-hoc data analytics. Because cluster initialisation can take a long time, users tend to have a cluster running all the time to avoid this start-up overhead every time a task/query is run. For large organisations, this is less of an issue because of having a more predictable workload from data science teams. However, for smaller organisations, usage is far more irregular and difficult to estimate a priori. Furthermore, a cluster of a fixed size could be under-utilised when it runs jobs which have different resource requirements. Since such a cluster does not scale elastically with workload, it will be under-utilised during executions of less resource-demanding jobs. Apart from that, a cluster's utilisation can also be affected by human errors such as forgetting to shut down a cluster over the

[1] AWS: Amazon Web Services

weekend [10] or misconfiguring a cluster such that it does not terminate after a job's completion.

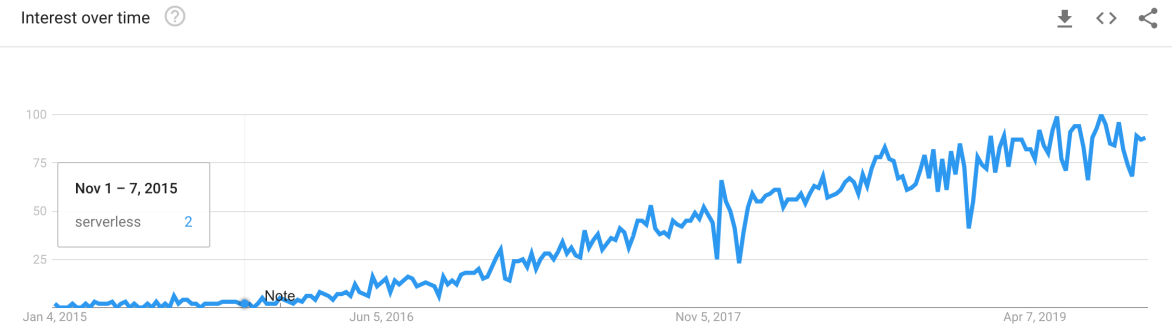


Figure 1.1: The popularity of searching the term "Serverless" on Google [16].

Recently, serverless computing has emerged [2] as shown by Figure 1.1. This new paradigm removes the complexity of using cloud as resource provisioning has been abstracted away. Serverless is a cloud computing architecture where the cloud provider runs the server and dynamically manages the allocation of resources for users. A prominent implementation of the serverless model is Function-as-a-Service (FaaS) [56]. The focus of this project will be on the FaaS model, hence the words "serverless" and "FaaS" will be used interchangeably. The reason why FaaS has become so popular is its ease of use. Users are required to just submit a function written in a high-level language and the serverless systems will take care of everything else such as instance selection, auto-scaling, deployment, fault-tolerance, etc. Moreover, serverless systems have high elasticity meaning that they can scale based on demand. Furthermore, serverless charges users with a pay-as-you-go model [42].

With these interesting characteristics, FaaS model is suitable for many types of applications [56] and presents a huge potential for tackling the aforementioned problems of cluster-based Big Data frameworks. If data processing jobs can be executed in a serverless architecture, users would no longer need to worry about provisioning and managing a cluster, and would only pay for the computation time when their jobs are running. Moreover, a serverless Big Data framework would be able to leverage the high elasticity of serverless computing to efficiently support jobs whose resource requirements vary significantly during their execution.

There have been attempts to develop Big Data frameworks that run in a serverless architecture. However, each of the implementations has certain limitations. Firstly, most of them still require at least one long-running server in a job execution. Secondly, some are implementations of simple serverless MapReduce that cannot express complex data processing jobs. Lastly, some implementations are not open-source, not maintained or have an extremely complicated setup.

1.2 Objectives

The goal of this project is to develop an open-source **framework** that deploys and executes a large-scale data processing job under a **fully serverless** architecture. It must:

1. Provide a simple way for users to write a data processing job and run it in a fully serverless architecture.
2. Be able to express all types of data applications including complex and large shuffling-intensive jobs.
3. Provide a way for users to execute a job locally, so that they can test the job without deploying to the cloud.
4. Provide a way for users to monitor the progress of any job executions.
5. Investigate the effect of different parameters such as the number of Lambda executors on the performance and cost of running different types of jobs in this framework.
6. Be thoroughly evaluated against cluster-based frameworks such as Hadoop on different jobs, to understand the cost and performance implications of turning a Hadoop job into serverless.

Chapter 2

Background

2.1 Cloud Computing

Cloud computing is on-demand delivery of computing services such as servers, storage, databases, networking, software, analytics and intelligence - over the Internet. Instead of owning and maintaining physical data storage and servers, users can easily access these services from a cloud provider [65, 33].

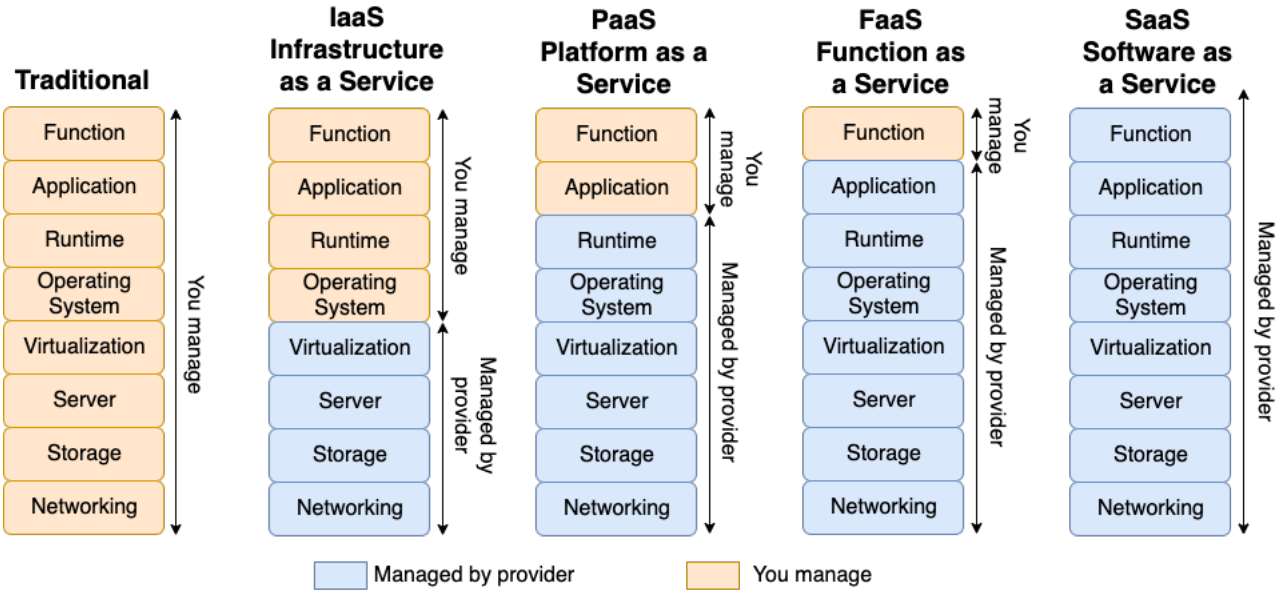


Figure 2.1: Main types of cloud computing models [11].

The four main types of cloud computing models and the traditional model are illustrated in Figure 2.1, along with the resource stacks that users and providers manage. Starting with the traditional model, you own the resources and therefore, you manage everything. Then the first type of cloud computing model is Infrastructure-as-a-Service (IaaS) where users rent immediately available IT infrastructure such as storage and servers. The next one is Platform-as-a-Service (PaaS), where a complete development and deployment environment is leased. System software such as OS, middleware,

DBMSs and libraries that assist in the development and deployment of cloud applications, are also provided to the users. After that, we have Function-as-a-Service (FaaS) which is a core implementation of the serverless model. This model is similar to PaaS, but FaaS only requires users to deploy essentially a single function, or part of an application. Moreover, in FaaS, capacity management such as on-demand autoscaling is all managed by the cloud providers. The last model is Software-as-a-Service (SaaS), where software applications are hosted and accessed by customers via the Internet [67].

2.2 Function-as-a-Service

As an implementation of the serverless model, Function-as-a-Service (FaaS) allows software developers to write and deploy individual ”functions” (written in a high-level language) without having to worry about any of the runtime details. All aspects of execution such as the allocation of servers is delegated to the cloud providers. Developers are also freed from the complex configuration required for IaaS and PaaS such as the level of autoscaling. Hence from a users' perspectives, there is no existence of servers and the overhead of server and infrastructure management is greatly reduced. This way, developers can spend more time writing application logic and less time worrying about servers.

In FaaS, the state of a function invocation cannot be carried to the next one. Hence functions for FaaS are stateless. Apart from that, they are event-driven. In other words, they are triggered by events such as user actions or updates in a storage. This form of computation encompasses many common tasks in cloud applications. A typical example is an image processing pipeline: When an image is uploaded to a website, it is stored in a S3 bucket, which later triggers a FaaS function to perform thumbnail generation [61]. The architecture of FaaS is shown in Figure 2.2. The platform manages a set of user-defined functions. When an event arrives via HTTP or from an event source, it is first pushed to the Event Queue waiting to be processed. Then the dispatcher determines which function the event belongs to and either selects an existing container of the function or start a new container with that function. Lastly, the container executes the function and returns the response to the user [47].

When a serverless function is triggered the first time or after a long time of inactivity, a container running that function is initialised [46]. This containerisation is what makes on-demand autoscaling possible. When the demand increases, new containers are spun up. The first time a function is invoked, it usually takes much longer than a typical request. This is due to the container having to be initialised, and load the runtime and all the necessary dependencies of the function. This long start-up time is known as ”cold start”. On the other hand, warm functions are functions which have running containers. Requests to warm functions take much shorter since they can be invoked without causing a container initialisation.

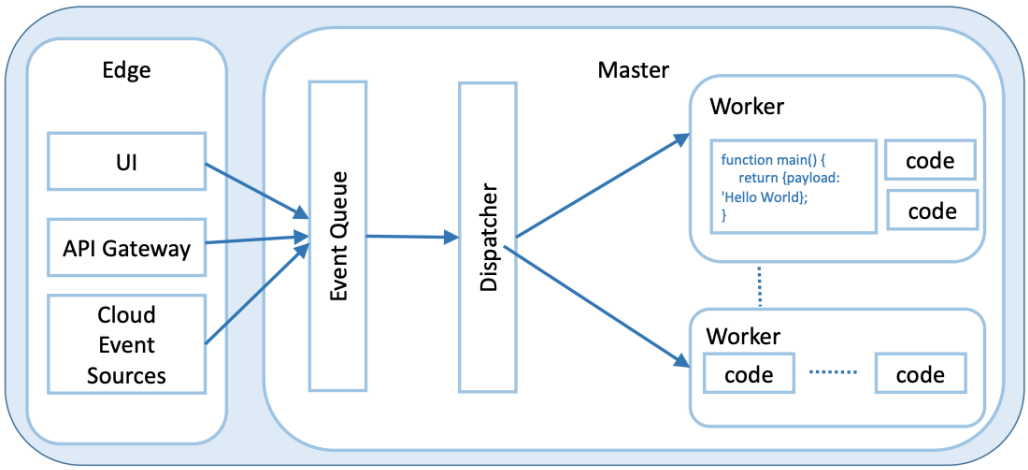


Figure 2.2: The high-level architecture of FaaS [47].

2.2.1 Limitations

Serverless platforms have greatly reduced operational and scaling costs. For example Adzic et al. present a 66 to 95% cost reduction in two industrial case studies [35], however it comes with some drawbacks. Common drawbacks include vendor lock-in which denotes the difficulty in migrating from one vendor to another since it will require major code change. Another drawback is that there are not well-defined ways of debugging and testing serverless systems when deployed or locally, and these platforms can have very slow tail latencies.

More importantly, the following FaaS's pitfalls bring extremely difficult challenges for implementing serverless data processing frameworks [54]:

- Serverless functions have limits on the execution time and the amount of memory that can be used during execution.
- There cannot be communication or sharing of data between any two function invocations.
- Serverless functions do not have states. Hence any states regarding the computation will have to be kept by other components.

2.2.2 AWS Lambda

AWS (Amazon Web Services) offering of FaaS is Lambda [45], which is what this project will be focused on and using. It supports a variety of languages such as Java, Go, Python and so on. Unfortunately, it comes with the following runtime environment limitations:

Resource	Limit
Function memory allocation	128 MB to 3,008 MB, in 64 MB increments
Function timeout	900 seconds (15 minutes)
Concurrent executions	1000 (Can be changed)
Invocation frequency	10 x concurrent executions limit (synchronous – all sources) Unlimited (asynchronous – AWS service sources)
Invocation payload	6 MB (synchronous) and 256 KB (asynchronous)
Deployment package size Disk Space (ephemeral)	50 MB (zipped, for direct upload) 512 MB

Figure 2.3: AWS Lambda Runtime Environment Limitations [36].

Note that there are different AWS Lambda resource models, hence some of the above limitations are expressed as ranges instead of absolute values.

To create a Lambda function, users specify a Lambda handler, which is the function that AWS Lambda executes when it is invoked [41]. The function signature of Python handler is shown in Listing 2.1 (it is similar in other languages).

Listing 2.1: Lambda handler function signature.

```
def handler_name(event, context):
    ...
    return some_value
```

Notice that a handler has two parameters:

- `event`: parameter that carries event data. In Python, this is usually of type dict. When it is the user who invokes the Lambda, the user gets to specify the content and structure of the event. However, when the Lambda is invoked by an AWS service, the event structure varies by service[1].

- `context`: parameter that records runtime information.

2.3 MapReduce

The main programming model for processing large datasets was introduced in 2004 by Google [51]. Since then, this model has been widely adopted by many modern well-known Big Data frameworks such as Apache Hadoop and Apache Spark, which will be explained in detail later. The key idea behind MapReduce is very simple but expressive such that many real-world tasks can be implemented using this model. It is composed of three stages:

[1]For details on the event structure of different services, see `https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html`

- **Map**: takes an input key/value pair and generates a set of intermediate key/value pairs.
- **Shuffle**: distributes all intermediate key/value pairs such that all values associated with the same intermediate key are grouped together and passed to a node that will perform the next phase (Reduce).
- **Reduce**: aggregates all the values associated with an intermediate key. It takes an intermediate key and a set of values for that key and produces possibly a smaller set of values. An optimisation applied in this step is the use of iterator to supply the intermediate values to this phase. This way, the Reduce step is able to handle lists of values that are too large to fit in memory.

Although there are three stages, to express a task, users only need to provide functions for the Map and Reduce stages. The function signatures of Map and Reduce are the following:

- **map**: (k1,v1) $\rightarrow$list(k2,v2)
- **reduce**: (k2, list(v2)) $\rightarrow$list(v2)

Note that the input pair domain of Map is different to the domain of intermediate and output pairs. On the other hand, intermediate and output key/value pairs have the same domain.

Intermediate results are persisted to local disks of the machines running the Map functions. Final results of MapReduce jobs are usually stored in a distributed file system. This allows one to pipe several MapReduce tasks together or run several iterations of a MapReduce task as we can feed the output of one MapReduce process into the input of another one [67].

Programs written in this functional style can benefit automatically from parallelism. Moreover, they are executed on a large cluster of commodity machines, which is another reason why a serverless architecture suits this kind of applications since serverless also consists of commodity machines. Users are exempt from things such as input data partitioning, scheduling of the program's execution across a set of machines, machine failure handling and management of inter-machine communications since they are all taken care by the run-time system. Porting this model to a serverless architecture will require changes to this run-time system.

2.3.1 Example

A typical example of MapReduce is word counting: count the occurrence of each word appearing in a large given set of documents. To express this task in MapReduce, users will provide code in a high-level language similar to the pseudocode shown in Figure 2.4:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Figure 2.4: Pseudocode for the word counting task in MapReduce [51].

A diagram showing the distributed dataflow of the task is illustrated in Figure 2.5. Note that in this diagram, there is another phase called splitting which involves partitioning the input data into a number of splits (as mentioned before, handled by the run-time system).

2.3.2 Google's implementation of MapReduce

The term MapReduce is also used to denote the internal implementation of MapReduce model in Google. Many different types of MapReduce implementations are possible. There is not a right implementation here. It all depends on the environment where the MapReduce implementation will run on.

Google's implementation targeted the main computing environment at the firm, which consists of a large number of commodity machines connected together with switched Ethernet. The execution flow of a MapReduce task in this implementation is shown in Figure 2.6. The following sequence of steps will happen when a job is executed (Note that the numbers in the list below correspond to the numbered labels in the figure) [51]:

1. Before anything else, the input data is partitioned into M splits by the MapReduce library in the user program. This partitioning automatically distributes the Map tasks across M mappers[2], with each one executing one of these M splits of input

[2]Mapper: Machine/Node running the map function.

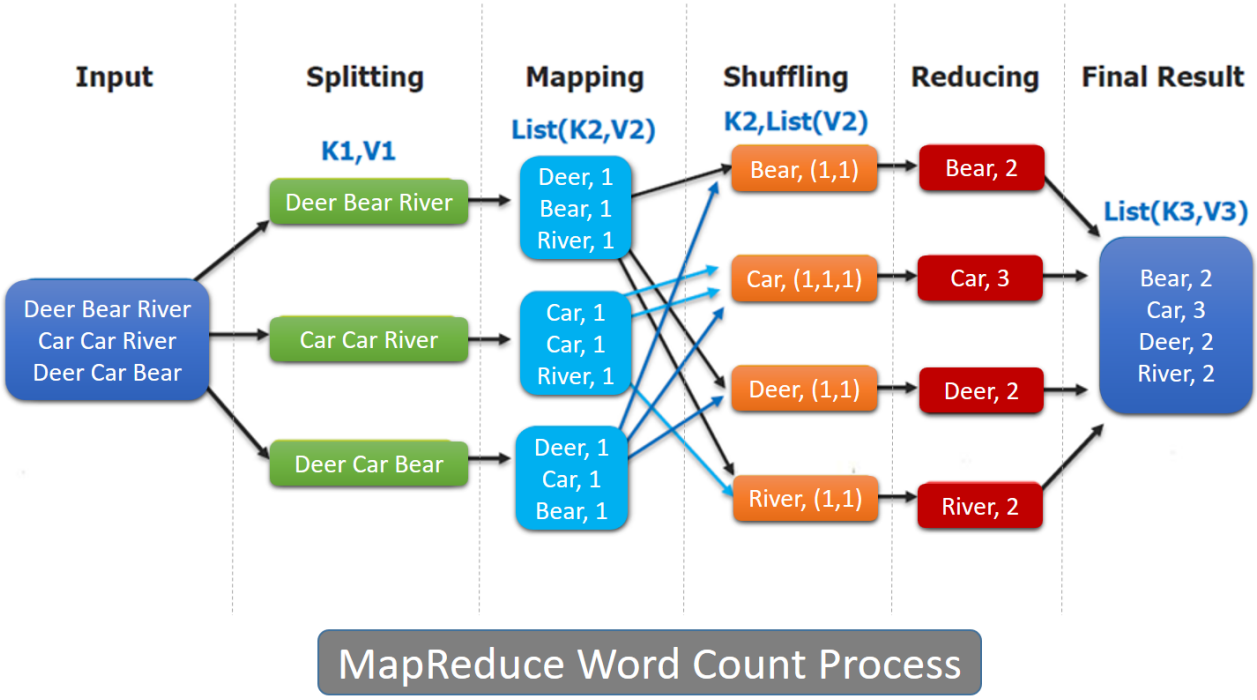


Figure 2.5: The dataflow of the word counting task with an example input [23].

data. Then copies of the user-provided program are started on the cluster of machines provisioned.

2. One copy of the program is special and the node executing it, is known as the master. The rest of the machines are known as workers. The master picks idle nodes and assigns each node a map or reduce task. Note that it is possible to assign a reduce task to a node that has executed a map task before.

3. A mapper first reads its corresponding input split. Then it executes the map function on each key/value pairs from the input split and produces intermediate pairs that are buffered in memory.

4. Periodically, the intermediate pairs buffered are written to the local disks. The disks are partitioned into R regions. An algorithm called partition function determines which region a pair belongs to. Both the number of partitions R and the partition algorithm are user's choice. The locations of the persisted intermediate pairs on local disk are informed to the master, which in turn, notifies the reducers[3].

5. When a reducer receives these locations, it uses the remote procedure call protocol to read the intermediate pairs from the local disks of the mappers. When all the data corresponding to one reducer has been read, the data is sorted according to the intermediate keys so that all values of the same key are grouped together.

[3]Reducer: Machine/Node running the reduce function

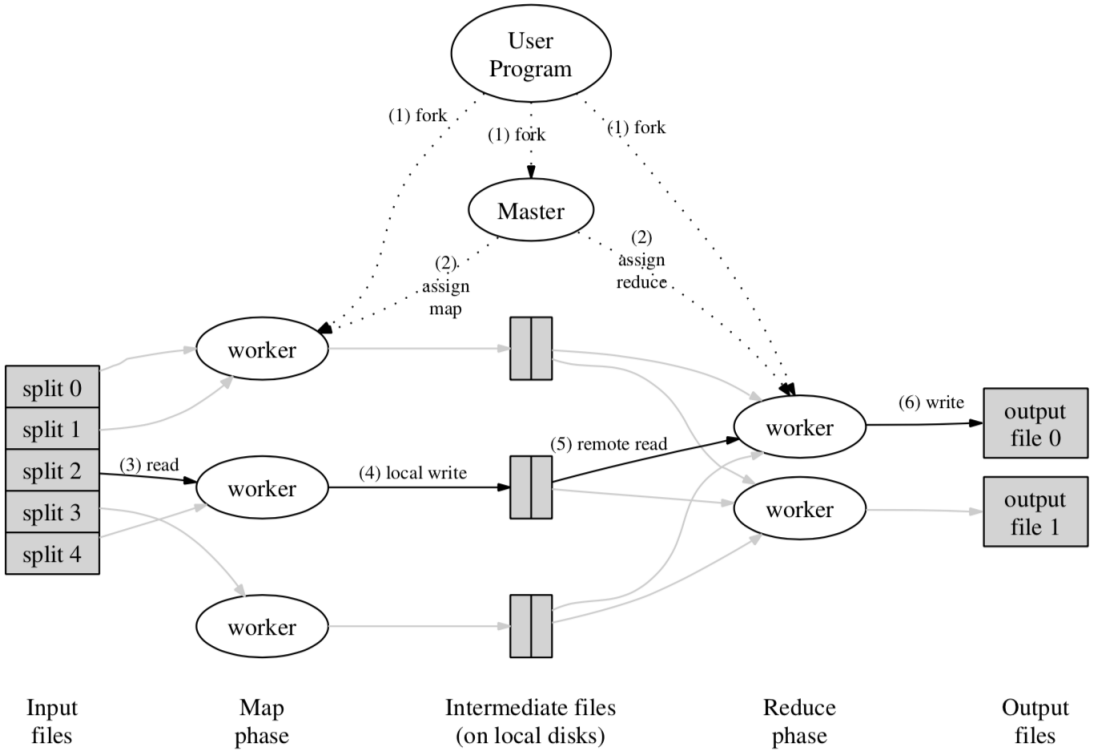


Figure 1: Execution overview

Figure 2.6: Execution Overview [51].

6. The reducer iterates through the sorted intermediate data and executes the Reduce function on each unique intermediate key with its set of values. The output file of this reducer is stored in a distributed file system. Therefore the final output of a MapReduce task consists of R output files, each one corresponding to one reducer. Users can choose to combine these R files into one if necessary.

7. Once all the map and reduce tasks have been completed, the user program is woken up by the master. The program control will return back to the user program from the MapReduce library.

2.4 Apache Hadoop

Apache Hadoop is an open-source framework that allows processing of large-scale data distributedly using cluster of low-level commodity hardware. It is an implementation of the MapReduce model, with some additional components [62, 64]:

- Hadoop Distributed File System (HDFS): a distributed file system that allows fast access to data across the cluster of machines. HDFS partitions the data into different chunks, which are distributed across different nodes in the cluster. It is the main storage used by Hadoop applications.

- Hadoop YARN: a technology that deals with cluster resource management and the scheduling of tasks across different cluster nodes.
- Hadoop MapReduce: open-source implementation of the MapReduce model to process large datasets in parallel.
- Hadoop Common: common utilities that are needed in order to use other Hadoop systems.

2.4.1 Example

In order to write a Hadoop job, users only need to specify two functions Map and Reduce along with the driver code. A Hadoop example is the word counting task. Due to the similarity of the Hadoop Map and Reduce functions with the MapReduce ones, the source code for these two functions will not be specified. What is new is the driver code. After the map and reduce functions are written, the main class where everything is combined together, is called the driver class. Its source code along with comments that explain the driver's responsibility [57, 55] is provided in Listing 2.2.

Listing 2.2: Hadoop driver code for the given example.

```
// Driver.java
public static void main(String[] args) throws Exception {
    // Create a configuration object for the job
    JobConf conf = new JobConf(WordCount.class);

    // Set the job name
    conf.setJobName("wordcount");

    // Specify the data type of the output key/value pair
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    // Specify the class names of the Map and Reduce functions
    conf.setMapperClass(Map.class);
    conf.setReducerClass(Reduce.class);

    // Specify the formats at which the input and output data will be.
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    // Set the file paths of the input and output.
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    // Run the job
    JobClient.runJob(conf);
}
```

2.5 Apache Spark

Even though MapReduce is powerful and expressive, it is very inefficient running two types of applications: iterative algorithms[4] and interactive data mining tools. In order to run iterative algorithms in MapReduce, iterations will have to be unrolled and a sequence of individual MapReduce processes will have to be submitted. However, then for each iteration, an IO cost is incurred due to reading data from a file system at the start of an iteration and writing it to a file system at the end of the iteration. For both types of applications, keeping data in memory will greatly improve the efficiency. This is how Apache Spark arose [71].

Spark was developed in 2009. Similar to MapReduce, it is a generalised framework for large-scale distributed data processing. However it has the additional property of caching and reusing data in memory across computations. In order to achieve this, it uses the concepts of Resilient Distributed Datasets (RDDs) and lineage graph.

2.5.1 RDD

An RDD is a programming abstraction that represents an immutable and partitioned collection of records. This abstraction also provides API for various data processing operations, materialisation of data, management of caching and partitioning of data. RDDs can only be created from data in stable storage or through operations on other RDDs. Operations on RDDs can be classified into two types: transformations and actions [44].

Transformations

Transformations are lazily-evaluated operations that generate a new RDD. When a transformation is invoked on an RDD, it is not executed immediately. In other words, the resulting RDD is not materialised immediately. Each RDD instead maintains its lineage which is information on how it is computed from other RDDs (its parents). Lineage is represented as a Direct Acyclic Graph (DAG)[5], hence it is referred as lineage graph. The nodes of a lineage graph are RDDs and the edges are transformations.

Some transformations consist of applying user-provided function to elements in one data partition, such as: `map`, `filter`, `flatMap`. Others apply user-defined aggregation function to elements from different partitions, such as: `groupby`, `sortBy`, `join`.

Actions

Actions are operations that trigger computation to materialise an RDD. Typically, this RDD represents the final output of a job. Actions tend to return a value to the program or save the materialised RDD's result to storage. Examples of actions are: `save`, `collect`, `count`.

[4]Iterative algorithms: algorithms that run repeatedly until certain condition is satisfied.
[5]Direct Acyclic Graph (DAG): graphs that have no cycles and each of its edges has a direction.

2.5.2 Lineage Graph (DAG)

An example of a lineage graph is shown in Figure 2.7. The standard workflow of any data processing job is as follows: read the data, apply some transformations on them and apply an action which materialises the result. Relationships between RDDs are created by transformations and there are two types:

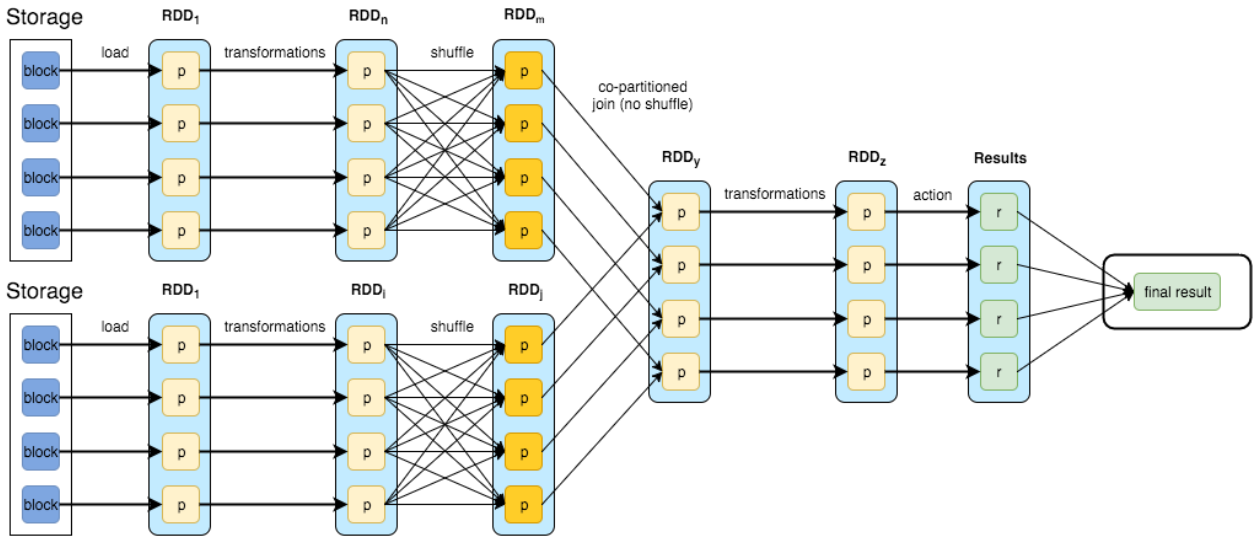


Figure 2.7: A lineage graph [44].

- Narrow dependencies: each partition of the parent RDD is used by at most one partition of the child RDD. Due to this property, the execution of a sequence of narrow-dependencies transformations can be pipelined on one cluster node without any data movement.
- Wide dependencies: each partition of the parent RDD is used by multiple partitions of the child RDD. Before executing a wide-dependency transformation, data from all parent partitions must be materialised and shuffled across nodes.

2.5.3 Execution Workflow

A high-level overview of Spark's execution workflow is shown in Figure 2.8. User program that defines a job using RDDs are turned into a lineage graph (DAG). This DAG is then split into stages of tasks by the DAGScheduler. Tasks that are connected together in the graph and do not require re-partitioning or shuffling of the data, are grouped into one stage. The stage boundaries indicate movement of data. Then these tasks are sent to the TaskScheduler which schedules each task to an executor. The executors execute the tasks and return the result to the user program.

DAGScheduler

DAGScheduler creates an execution DAG or physical execution plan for a job by slitting its lineage graph (DAG) into different stages. The stage boundaries indicate the

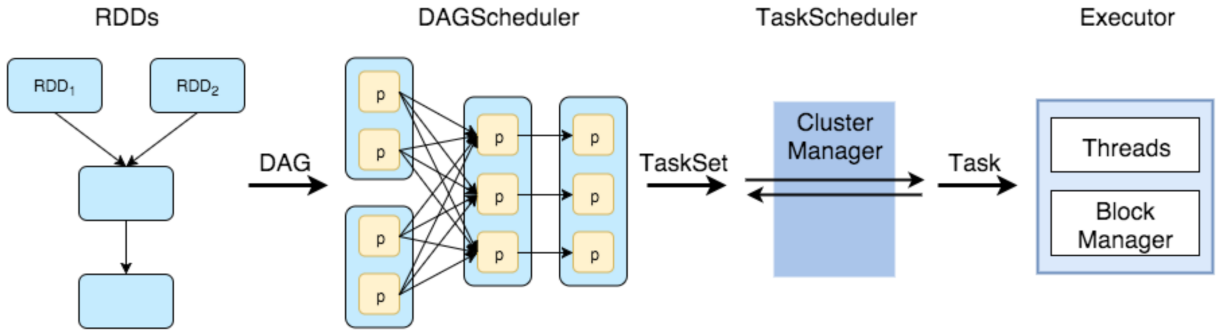


Figure 2.8: Spark Execution Workflow [44].

requirement of data shuffling. Neighbouring narrow-dependency transformations are grouped into the same stage. Each of the transformations inside a stage is a single task and can be pipelined. Wide-dependency transformations have shuffling dependencies on other stages. Hence they define the boundaries of the stages. Figure 2.9 shows how the lineage graph in Figure 2.7 would look like after being broken down into stages. Apart from that, DAGScheduler also determines the best locations to run each task.

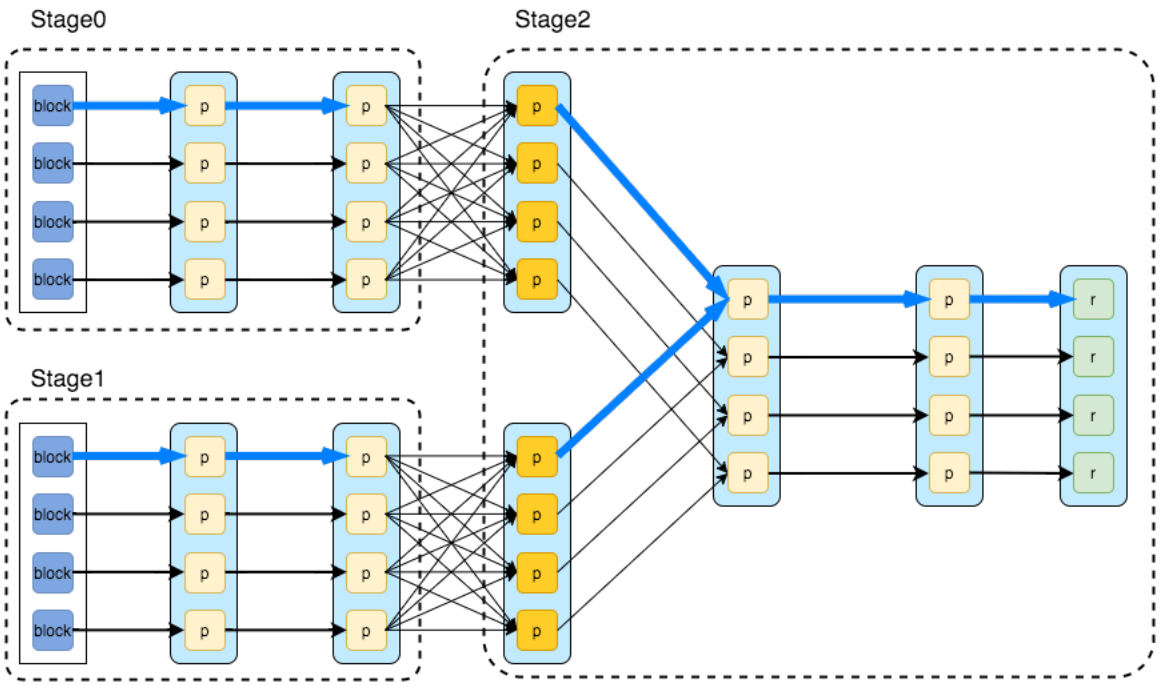


Figure 2.9: Lineage graph split into stages [44].

TaskScheduler

For each stage, the DAGScheduler submits sets of tasks (known as TaskSets) to the TaskScheduler, which in turn, sends these tasks to the cluster and handles any failures by retrying the tasks. Note that one stage of tasks can be transformed into multiple

TaskSets. The TaskScheduler coordinates with a component called SchedulerBackend to do its job.

Executors

Tasks are then sent to specific executors according to data locality and resource constraints [28]. Executors are Spark processes that run inside worker or slave nodes and are responsible for task execution. Execution of task involves performing data processing, reading from and writing data to external storage.

SchedulerBackend

Spark has a pluggable backend mechanism called SchedulerBackend to support various cluster managers such as YARN [29]. Each of these cluster managers differs in its custom task scheduling mode and resource offer mechanism, and Spark abstracts these differences via the interface SchedulerBackend. Implementations of SchedulerBackend are provided by cluster managers and offer resources where tasks will be scheduled on. There are different types of SchedulerBackend which differ by the location that the cluster resources are from [5]:

- Local mode: the driver[6] and executors run as one process in one machine.
- Local cluster mode: driver and executors run as different processes in one machine.
- Cluster mode: The driver and each executor runs in a different machine.

Spark provides some default implementations of SchedulerBackend that correspond to different cluster managers: YARN, Standalone, Mesos and K8s.

2.5.4 Example

An example of a Spark application is word counting. It is very simple to write the code for such a task, which is shown in Listing 2.3. For monitoring and instrumentation, Spark provides a web UI that shows useful information about jobs. For instance, it shows the execution DAG (illustrated in Figure 2.10) of the word counting job.

Listing 2.3: Spark code for the word counting example [4].

```
public class SparkAppMain {

    public static void main(String[] args) {

        String fileName = args[0];

        // Create the configuration object for the application
```

[6]The driver is the program where users declare the transformations and actions on RDDs and it runs the main() function of the Spark application.

```
        SparkConf sparkConf = new
            SparkConf().setMaster("local").setAppName("Word Counter");

        // SparkContext represents the connection to a Spark cluster
        JavaSparkContext sparkContext = new JavaSparkContext(sparkConf);

        JavaRDD<String> inputFile = sparkContext.textFile(fileName);

        JavaRDD<String> wordsFromFile = inputFile.flatMap(content ->
            Arrays.asList(content.split(" ")));

        JavaPairRDD<String, Integer> countData = wordsFromFile.mapToPair(t
            -> new Tuple2(t, 1)).reduceByKey((x, y) -> (int) x + (int) y);

        countData.saveAsTextFile("CountData");
    }
}
```

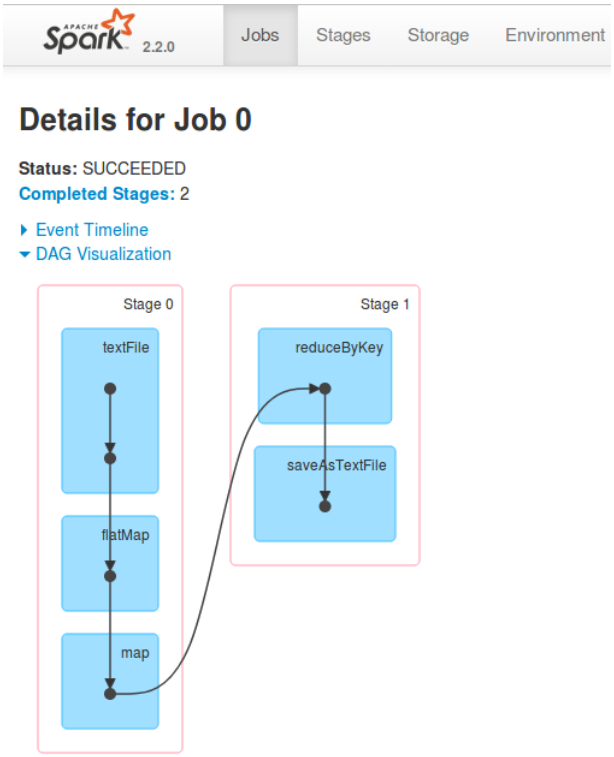


Figure 2.10: Execution DAG of the word counting job [70].

2.6 Serverless Big Data frameworks

Even though Big Data frameworks such as Apache Hadoop and Apache Spark work well, as mentioned in the introduction, there are problems that come with running

them on dedicated clusters of machines. Two main issues are [59]:

- Complex cluster management and configuration.
- Higher cluster cost than necessary due to also paying for the idle times of machines.

Serverless presents a great potential in solving these two issues. By executing Big Data frameworks on a serverless architecture, users would no longer have to worry about provisioning and managing a cluster and would only pay for the computation time that their job spends running. By abstracting away the infrastructure management, serverless would simplify the process of defining data processing jobs for users. Users would be able to spend more time focusing on modelling data processing problems and deriving value from data rather than learning how to use Spark or Hadoop [3]. Moreover, for ad-hoc analytics and exploratory data analysis, a serverless Big Data framework would make more sense since it would not require cluster initialisation for each query/task. Apart from that, the elasticity of serverless functions would allow a job execution to scale based on its demand.

As a result, there have been several attempts at building serverless Big Data frameworks. However, the majority come with certain limitations. Each of these solutions and its drawbacks are explained in detail in later section.

2.6.1 Main challenges

Running Big Data frameworks on a serverless architecture seems like the perfect solution. Unfortunately, there are several challenges in developing them due to serverless's properties [54]:

- Resource limitations: FaaS comes with limitations on execution time, memory size and so on. AWS Lambda's limitations have been mentioned in the section of AWS Lambda. Limit on the execution time restricts how long a task can be. For example, a map task of one partition that takes 20 minutes, cannot be executed directly in Lambda that has an execution time limit of 15 minutes. Other limitations such as memory size limit make the initial split of input data more complicated since the framework will have to make sure that the size of an input partition does not exceed this memory limit.
- Stateless: Both MapReduce and Spark require maintaining state of the execution such as which tasks have been executed so far. Unfortunately, FaaS is stateless which means that information about execution's state needs to be stored in an external storage.
- No communication between invocations [58]: At the shuffle stages, data are moved from nodes in one stage to nodes in the next stage. As no communication is allowed between two Lambda invocations, there is no way of transferring data from one to another. Therefore, a fast-access storage that stores all the intermediate data is necessary.

- Cold start time: FaaS has cold start time during which it creates a container and loads the function to be executed with its dependencies. This cold start time might detriment the performance of a serverless data framework.

2.6.2 Existing work

AWS Serverless MapReduce

AWS attempted to build a serverless MapReduce [3, 6] that makes use of AWS Lambda in conjunction with AWS S3. It has an architecture that consists of three main Lambda functions:

- Mapper: executes the map function.
- Reducer: executes the reduce function.
- Coordinator: maintains state information of the execution. This information is stored in S3 since the coordinator itself is a Lambda. The job of the coordinator is to detect the termination of the map stage and schedule the reduce stage.

The execution workflow of this framework is shown in Figure 2.11. A detailed explanation of each step of this workflow is also provided (Note that the numbers in the list below correspond to the numbered labels in the figure):

1. The job execution is initiated by the invocation of the driver script (usually in a local computer). This script loads the configuration file which contains the file paths of the mapper and reducer, and S3[7] input bucket name. Then the driver function locates the input objects stored in S3. The objects' keys, i.e., their S3 file paths are aggregated into batches, which are then passed to the mappers. The size of the batch can be obtained through a simple heuristic technique that tries to maximise parallelism while not exceeding the mapper memory size limit. The driver also creates the mapper (with user-provided map function), reducer (with user-provided reduce function) and coordinator Lambdas.
2. The driver invokes one mapper Lambda function per batch. Keys in one batch correspond to the input partition that a mapper needs to load and process.
3. A temporary workspace for the current job is created and stored as a folder in S3. Whenever a mapper finishes its task, it writes its output to this S3 folder. This event triggers and invokes the coordinator Lambda function.
4. After all the mappers terminate, the coordinator Lambda uses the aforementioned mechanism, to create batches for the reducers and invoke one reducer per batch.
5. When all reducers of current stage finish running, the coordinator is informed through the S3 event source and creates subsequent stages of reducers until a single reduced output is produced.

[7] S3: Amazon Simple Storage Service is an object storage. It can also be considered as a key-value storage since S3 is 'similar' to a file system where objects are stored as 'files'.

6. The final single reduced output is written back to S3.

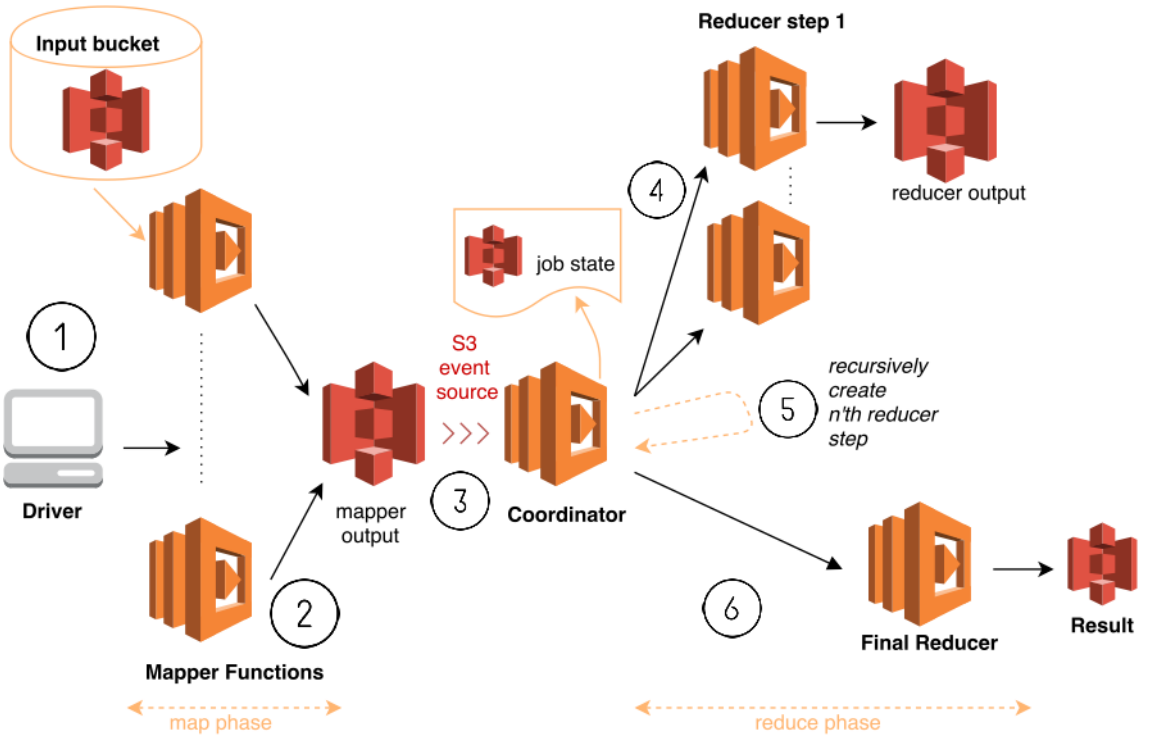


Figure 2.11: Execution workflow of AWS Serverless MapReduce [3].

Because this solution does not have the shuffle stage, it has to recursively reduce the intermediate files until a single reduced output is produced. This is unscalable and inefficient as for each intermediate key, its values are not reduced all at once, but a few in each reduce phase. Additionally, this requires that the output of a job fits in the memory of a single Lambda function (the only reducer in the final reduce phase). This constraint limits the amount of output data that can be produced by this solution. Furthermore, we believe that this solution is more of a prototype since it is not provided in the form of a library. Users are required to download the project from GitHub and then make changes directly to the map and reduce functions in the framework's source code. Apart from that, it only works for simple data processing jobs since it does not support chaining of map and reduce stages. Therefore, jobs can only be composed of a map phase followed by a reduce one.

PyWren

PyWren is another project of serverless data framework. Although the paper [59] does not explicitly specify whether it is serverless Spark or MapReduce, it can be deduced that this solution is the latter. Unlike other serverless data frameworks, PyWren is a simple Python API that allows users to submit map and reduce tasks that are executed as stateless functions. Its API mirrors the Python API for parallel processing, hence it can be easily integrated with existing libraries for data processing and visualisation.

Three main components (depicted in Figure 2.12) are needed to build such a system: an execution runtime, a scheduler and an external storage. Map or reduce functions (expressed as stateless functions) along with the required runtime dependencies are submitted to the scheduler. Once the scheduler decides where the function should be executed, a container with the function is initiated. As output computed by the function cannot be maintained between invocations, a remote storage is needed to maintain such data.

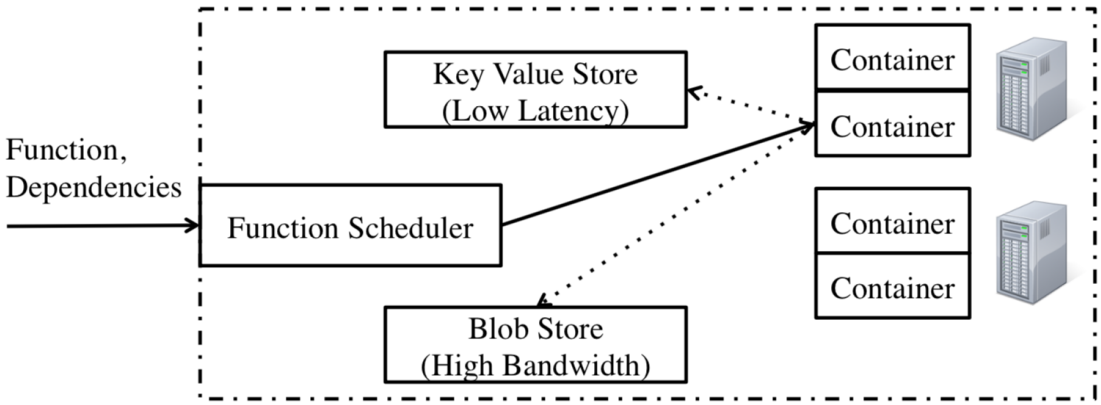


Figure 2.12: High-level architecture of PyWren [59].

Its execution workflow is as follows:

1. PyWren first serialises a user-provided Python function (for example, a map function) and the associated data using cloudpickle [9]. Then the serialised function and data are stored in S3 with globally unique keys.

2. A common Lambda is invoked and loads both the function and the relevant data to be processed from S3. Note that PyWren only uses a single general Lambda that loads the serialised function to be executed from S3 and executes it. This way, one registered Lambda function can be reused to execute different user-written Python functions and mitigate the problems of cold start. Moreover, it also allows PyWren to run functions that exceed Lambda's code size limit.

3. The common Lambda executes the serialised function and produces an output that is serialised and stored into S3 at a pre-specified key. The existence of such key will signal job completion.

The paper claimed that PyWren has been used for several types of applications such as computational imaging, scientific instrument design, solar physics, and object recognition. However, one of its drawbacks is that large shuffle intensive workloads is not easily supported by this solution. For these applications, the throughput of the storage is the major bottleneck. Apart from that, with the current implementation, a function takes about 20 to 30 seconds to launch (cold start). Another limitation with PyWren

is that the function scheduler runs on the developer's machine, hence it is not fully serverless. This makes it unsuitable for running jobs on a schedule and the function scheduler is subject to becoming a bottleneck of the entire system. Moreover, PyWren does not support chaining of multiple map and reduce functions. As an interesting fact, the paper of PyWren won the best vision paper award at the ACM Symposium on Cloud Computing event. This shows how much potential serverless has for Big Data frameworks.

Corral

Corral [20] is a serverless data processing framework for MapReduce jobs written in Go. Unlike AWS Serverless MapReduce, Corral does not have the Coordinator component that schedules the reduce phase when the map phase finishes. Instead, this responsibility is given to the Driver, which runs on a local machine. Figure 2.13 shows the timeline of a job execution on Corral.

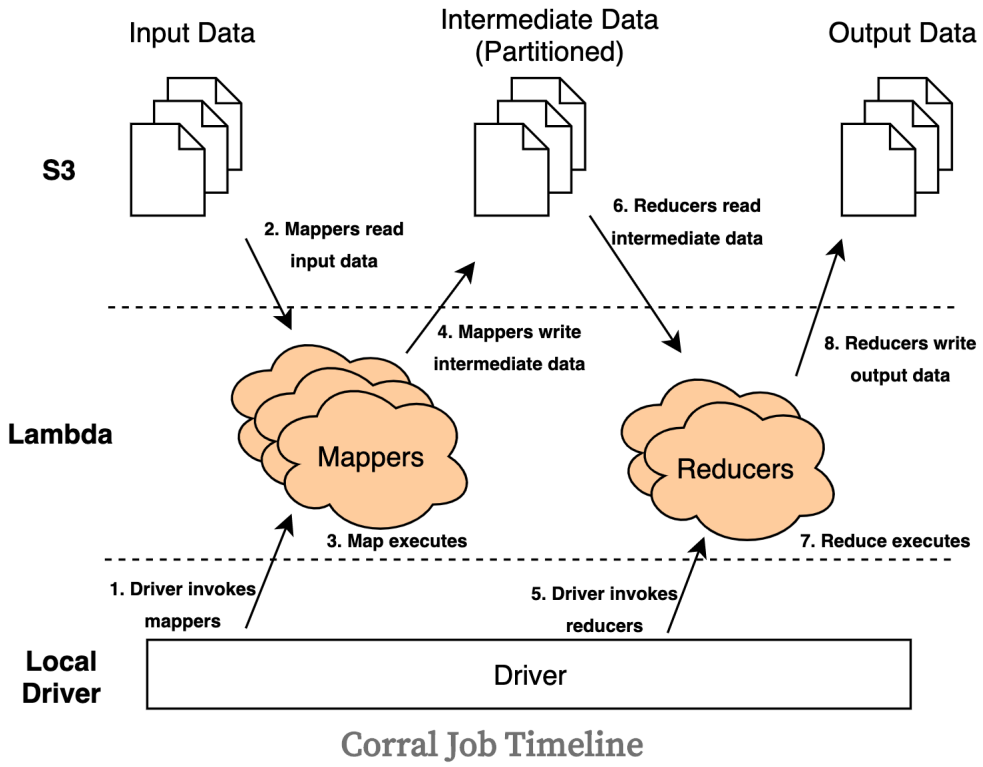


Figure 2.13: Job timeline of Corral [20].

Notice that Corral only has one reduce phase with n reducers, unlike AWS Serverless MapReduce. This optimisation is possible because Corral has an additional phase between the map and reduce ones: shuffling. Shuffling ensures that all intermediate values associated with the same intermediate key will be grouped together and pro-

cessed by the same reducer.

Corral adopts S3 as the back-end for a stateless shuffle and this S3 bucket stores intermediate data that will be later processed by the reducers. It uses the prefixes of S3 file/object names to distinguish between files that belong to different partition bins. During the map stage, key/value pairs emitted are written to intermediate files. Keys are partitioned into n bins where n is the number of reducers and each bin is processed by one reducer. Hence one mapper can write as many as n different intermediate files, each representing one bin and containing keys that belong to that bin. Each intermediate file is named in the following format: map-binX-Y where X is the bin's ID between 0 and (n - 1) and Y is the mapper's ID between 0 and (number of mappers - 1). An example of using this shuffling technique is shown in Figure 2.14.

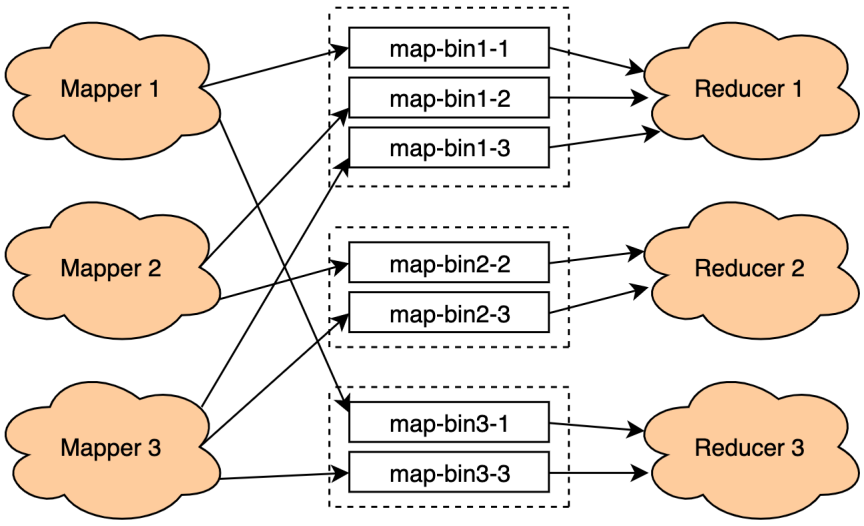


Figure 2.14: Corral shuffling example [20].

Corral is suitable for data-intensive but computational inexpensive tasks, such as ETL jobs. However it does have limitations. Firstly, it can only process standard simple map reduce jobs. Chaining of map and reduce phases are not supported, hence it cannot express complex MapReduce jobs. Apart from that, since the driver runs on a local machine, it is subject to becoming a bottleneck of the entire system since it has to coordinate the transition from the map stage to the reduce stage. Additionally, the driver has to be available during the entire job execution.

Flint

Unlike previously mentioned solutions, Flint [61, 60] is a Spark execution engine that uses a serverless architecture. It allows users to use PySpark as before. Moreover, it

removes the need for a Spark cluster and offers a pure pay-as-you-go cost model.

As illustrated in Figure 2.15, Flint's overall architecture is composed of 3 main AWS services, namely Lambda, S3 and SQS. AWS Lambda is used to execute Spark tasks. AWS SQS is used to store intermediate data and handles the data shuffling that is required by many transformations. AWS S3 is used to store partitions of the input and output data.

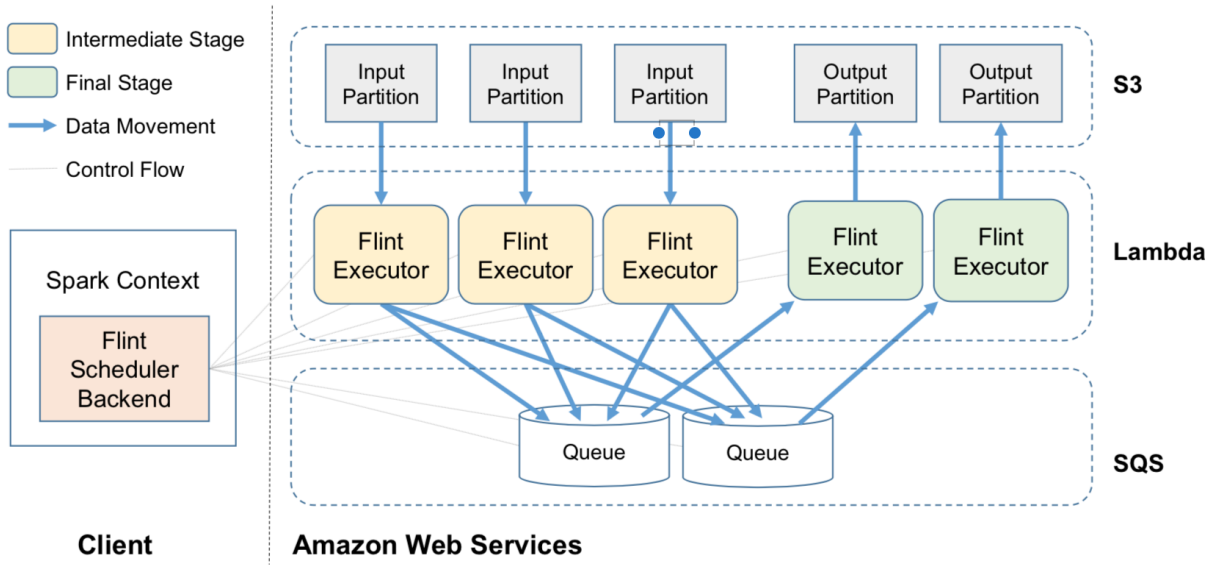


Figure 2.15: High-level architecture of Flint [61].

The execution workflow of Flint is as follows (you will notice that most of the steps remain the same as standard Spark's execution):

1. When a Spark job is submitted, the lineage graph of the job is converted into a physical execution plan (DAG with stages) by the DAGScheduler. This plan is then sent to the TaskScheduler.

2. The TaskScheduler collaborates with the SchedulerBackend of a cluster manager, to schedule the execution of tasks in TaskSets. As there was not an implementation of SchedulerBackend for serverless Spark, Flint SchedulerBackend (is referred as "scheduler" hereafter) was created. Its job is to coordinate Flint executors to execute a specific physical plan. A Flint executor is a process running inside an AWS Lambda function that executes tasks in the physical plan. It is also the scheduler's responsibility to register a common Lambda function if it does not exist.

3. A pool of threads is maintained by the scheduler for the management of Lambda executions. The size of the pool determines how many concurrent Lambda invocations there can be at one point in time. Every time there is an idle thread in this pool, tasks will be requested from TaskScheduler by the scheduler. Then

for each task, the scheduler extracts and serialises various information needed for the task execution by Flint executors. This information includes input location, output location, serialised execution code and metadata about the relation between this task and the entire physical plan.

4. Once the serialisation is complete, idle threads asynchronously invoke the common Lambda function with the serialised task as part of the request.

5. The Flint executor executes the task and once the task has been completed, it sends back the response to the scheduler. When all tasks of the current stage complete, requests for the next stage's tasks are made to the Lambda function, repeating until the entire physical plan has been executed.

A special case that has not been mentioned in the execution workflow is shuffling. Shuffling is automatically handled by this solution since it forms part of the physical plan that is created by Spark. Flint executors simply execute the tasks and are fully unaware of the actual underlying RDD transformations. For instance, they do not know whether the shuffling task belongs to a join or reduceByKey transformation.

This solution also leverages some interesting approaches to address some of the Lambda's limitations:

- Cold start: long cold start time was claimed to be addressed by choosing to implement Flint executors in Python. The reason is that Lambdas of compile languages have larger deployment packages compared to Python Lambdas due to requiring many more dependencies.

- Execution time limit: executors are chained. If a running executor is reaching its time limit, then it stops fetching new input records from its partition (Flint executors stream the input data from an iterator). Information about the current state such as how much of the input split has been read, is serialised and returned to the scheduler. The scheduler then launches a new Flint executor to continue processing the unfinished input split from where the previous invocation left off.

- Payload size limit: as the code of a task is passed in as part of the Lambda request payload, it is important to overcome this limit. Flint's workaround for this issue is uploading the serialised code to S3 and then, the scheduler can direct executors to download the task code from S3.

Flint leverages AWS SQS to deal with the bottleneck of storage throughput for large shuffling intensive jobs. However, from the paper, it can be deduced that it does not completely solve the problem as the standard Spark's execution time is significantly lower than Flint for this kind of applications. Apart from that, due to extensive usage of AWS SQS, the cost shoots up. For certain applications, the cost of running them using Flint is twice the cost of using standard Spark (This is comparing the query cost). Furthermore, Flint's Spark Context is required to run on a long-running server (such as the developer's machine), hence it is not fully serverless, which makes it unsuitable for running jobs on a schedule.

Qubole

Qubole [8] is a company that offers multi-cloud data platform for data processing. It has also endeavoured to build a serverless Spark on AWS Lambda. Like Flint, the overall execution workflow remains the same as standard Spark. The only two key architectural changes are [25, 26]:

- Spark executors run inside an AWS Lambda function.
- Shuffle operations are performed via external storage as a workaround to avoid memory size limit of Lambda and inter-communication between Lambda invocations.

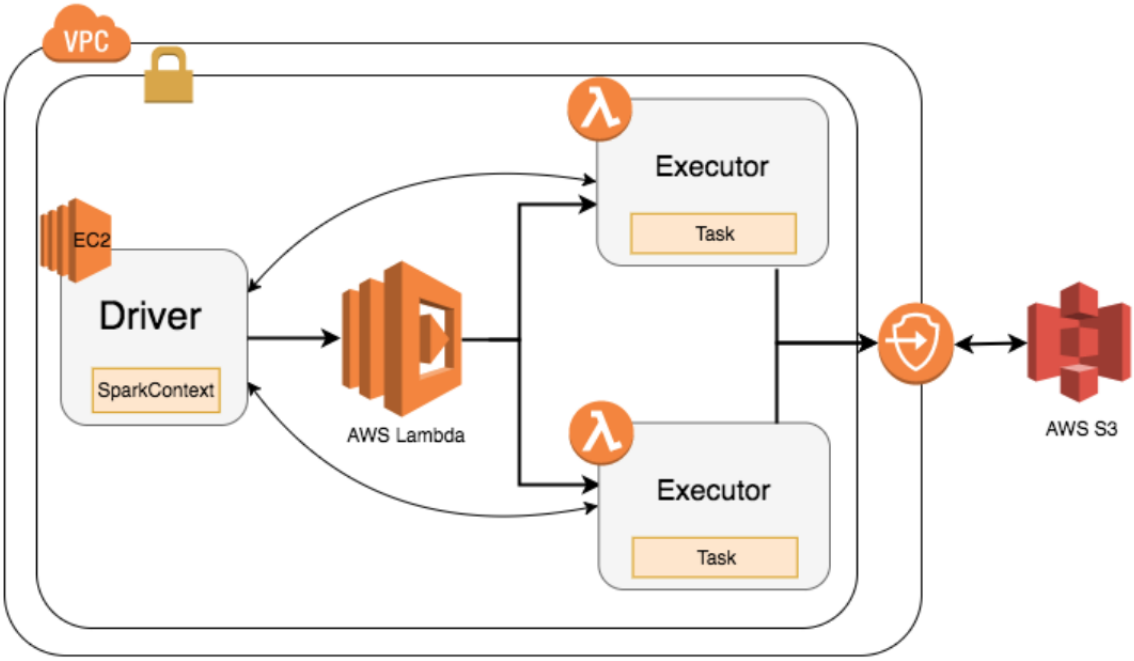


Figure 2.16: Qubole's Spark architecture [25].

The new Spark's architecture is illustrated in Figure 2.16. The steps to run executors on Lambda are the following:

1. Launch a general AWS Lambda with a barebones Python Lambda function.
2. After the launch, this Python function bootstraps Lambda runtime by downloading Spark libraries from S3 and starts the Spark Executor by executing a Java command passed in as part of the Lambda request payload.
3. Then, this Executor joins the Spark application by sending heartbeats back to the Spark Driver.

Apart from making changes to Executor, a new Spark SchedulerBackend had to be built as there was no off-the-shelf implementation of SchedulerBackend for a serverless architecture. This new SchedulerBackend is called LambdaSchedulerBackend. It makes

a request to AWS Lambda to get a Lambda invocation, whenever the Spark driver requests for more resources. Once the invocation is available, the Executor is initiated as described previously. As soon as an executor's time limit reaches the end, the auto-scaling component of Spark asks for new executors and new requests are made to AWS Lambda. Apart from that, in order to reduce the chance of task failures, Spark scheduler stops scheduling tasks to an executor whose execution time is close to expiry.

Intermediate data shuffling is done via AWS S3. Mappers write their outputs to S3 with specific prefixes as file paths in a directory layout scheme. This allows executors in the next stage to easily select which files (partitions of data) they want to read from S3. Using this approach, no external shuffling service is required.

There are some differences between Qubole and Flint. Firstly, Qubole tries to port the existing Spark executor infrastructure onto AWS Lambda, whereas Flint implemented serverless executor from scratch. Hence, Qubole is more limited in that it cannot be easily optimised for the unique execution environment of Lambda. Moreover, Qubole uses S3 for intermediate data shuffling as opposed to SQS, used by Flint. Unfortunately, there have not been any performance or cost comparisons being made between these two tools.

Qubole's drawbacks include high cost and long running time. This is mainly due to occasional S3 throttling (S3 does not allow more than X concurrent writes to a bucket), expiring Lambda execution duration causing task to fail and to be retried. The long running time is also partially due to the cold start of Lambda, which is not addressed in this solution. Like Flint, Qubole's Spark Context is required to run on a long-running server (such as the developer's machine), which makes it inappropriate for running jobs on a schedule.

Summary

There are other implementations of serverless Big Data frameworks that are not mentioned here such as IBMPyWren, gg and Locus. Most of them share similar approaches as previously described solutions. For instance, every solution uses a remote storage to keep intermediate data. To summarise, there has been extensive related work been done. However, these implementations have different limitations. Some are very simple serverless MapReduce frameworks which cannot express complex data processing jobs. Others are serverless Spark implementations that are not open-sourced (Flint), not maintained or have an extremely complicated setup (Qubole due to poor documentation and no maintenance). Apart from that, most of them are not fully serverless and still require at least one long-running server in a job execution. Furthermore, in implementations such as Flint and Qubole, they try to incorporate Lambda into architectures that are designed to run with persistent long-running servers. In these implementations, a Lambda executor is required to replicate the behaviour of an executor in a long-running server. Because of this, additional overhead can be introduced and Lambda is not fully leveraged in these solutions.

2.7 AWS Storage Services

2.7.1 AWS S3

AWS Simple Storage Service (S3) is an object storage service that offers high scalability, data availability, security, performance and low cost. AWS S3 is designed for 99.999999999% (11 9's) of durability. Sometimes, S3 is mistaken as being a file system that has a hierarchical structure. However, it is a flat structure of objects. Objects are stored in data storage containers called buckets. Buckets and objects are addressed by keys not file paths. These key names often contain the character '/', hence the confusion [31].

S3's high availability is achieved by replicating data across multiple servers within AWS data centres. If a PUT request is successful, your data is safely stored. However, there might be delays on propagating information about the changes across different S3 servers. For **overwrite PUT and DELETE requests**, S3 guarantees eventual consistency. This means that you always get consistent results, i.e., either the old or new state of an object and never the intermediary, non-complete or corrupted version of an object. For **PUTS of new objects** in a S3 bucket in all regions, S3 provides read-after-write consistency with a caveat. In other words, any reads (HEAD/GET) of an object after its PUT request always returns the object [43]. The caveat is that if a read request is made to an object before its creation and then a PUT request of this new object is issued, a subsequent read might not return the object due to eventual consistency. Listing 2.4 shows the scenario where this caveat occurs:

Listing 2.4: S3 read-after-write consistency caveat.

```
GET /prefix/object.jpg 404
PUT /prefix/object.jpg 200
GET /prefix/object.jpg 404
```

A GET request is made to the object before its creation. At this point, since the object does not exist, a 404 Not Found response is received. Then a PUT request of the new object is issued and a response of 200 is received. The final operation is another GET request for the object, but a 404 Not Found is received again. In this scenario, S3 has eventual consistency and the 404 response for the first GET is returned until the PUT request has fully propagated [27].

2.7.2 AWS DynamoDB

AWS DynamoDB is a NoSQL key-value and document database. It is suitable for workloads that require predictable read and write performances with any amount of data. It can scale up and down to support the specified read and write capacity in provisioned capacity mode or can autoscale in on-demand mode. In DynamoDB, you can create database tables to store data. Tables consist of Items (rows) and Items consist of Attributes (columns). Note that tables do not have a schema since DynamoDB is a NoSQL database. This means that every item in a table could have different attributes

(columns) [14].

DynamoDB tables are partitioned. It automatically creates partitions of a table when it grows too large or when there exists a hot partition (a partition that is read/write much more often than other partitions). It always try to evenly split the reads and writes across partitions to maintain the predictable read and write performances. A typical latency of an operation in DynamoDB is in the 10ms–20ms range [1].

It supports Eventually Consistent Reads (default) and Strongly Consistent Reads [38]. The level of consistency reads can be set on a per-call basis. When reading data from a table with eventual consistency, the response might not correspond to the latest write operation. Even though this type of reads can include stale data, it has a lower latency compared to Strongly Consistent Reads. Strongly Consistent Reads will always read from the leader partition since it always has an up-to-date copy. Data read will never be inconsistent, but might have a higher latency.

Even though DynamoDB tables do not have a schema, it is required for any table to have a common primary key between all items (rows) [14]. The primary keys determine how your data will be partitioned and in which partitions each item will be stored. The Key schema can be composed of two keys [15]:

- Partition Key: also known as the HASH key. This key serves as input to a hash function and the output from this function is used to determine the partition in which the item will be stored.
- Sort Key: also known as the RANGE key. When multiple items in a table have the same partition key, their sort keys are used to store the items in sorted order by the sort key values.

There are two types of primary keys:

- Single Primary Key: Using only a partition key. With this key type, no two items can have the same partition key value.
- Composite Primary Key: Using both a partition key and a sort key. The combined partition key and sort key must be unique for each item.

Lastly, DynamoDB supports transactions and atomic operations such as atomic updates [32]. This can be useful to avoid race conditions and ensure data consistency and integrity in a distributed setting.

Chapter 3

ServerlessMR API

This project presents **ServerlessMR**, a Big Data analytics framework that leverages serverless functions to run MapReduce jobs in a completely serverless architecture. ServerlessMR can be seen as a light-weight alternative to Hadoop MapReduce. Currently it uses AWS serverless functions - AWS Lambda.

3.1 Client API

The framework ServerlessMR provides a similar API as other standard MapReduce frameworks. An object of the class ServerlessMR, used for any interaction with the framework, can be instantiated by calling `ServerlessMR()`. In the framework, a job can be composed of several pipelines, with outputs of various pipelines being the input of another one. A pipeline is defined as a sequence of map and reduce stages that are executed one after another, with output of one stage being the input of the next one. The class ServerlessMR offers the following API methods for users to create a data processing job:

- `map(map_function)`: appends a map stage with the given map function to the current pipeline.

- `combine(combiner_function)`: this method can be invoked following a `map` API method call if there is a `reduce` after it. The default combiner function[1] of a shuffling stage is the subsequent reduce function, but a custom combiner function for a particular shuffling stage can be provided using this API method.

- `shuffle(partition_function)`: this method can be invoked following a `map` or a `combine` method call if there is a `reduce` after it. If this method is specified, then in the shuffling for subsequent reduce stage, the provided partition function would be used instead of the default partition function.

[1] In the map stage, after the map function is applied to the input data, each mapper can then apply a combiner function to the output key-value pair produced by itself. This combiner function usually aggregates the values of each key and the outputs of this function are stored as the intermediate data instead.

- `reduce(reduce_function, num_of_reducers)`: appends a reduce stage with the given reduce function to the current pipeline. The number of reducers to be scheduled for this stage is specified by the other parameter `num_of_reducers`. Note that a `reduce` method call must follow a `map/combine/shuffle` function call.
- `finish()`: returns an unique identifier for the current pipeline, known as pipeline id and starts the definition of a new pipeline. Invoking this method signals the framework that the current pipeline's definition is finalised. Note that it does not start the execution of the current pipeline.
- `merge([pipeline1, pipeline2, ..., pipeline n])`: given a list of pipeline ids, this method instructs the current pipeline to combine the outputs of the corresponding pipelines. These combined outputs will then be sent into the current pipeline as input.
- `config(pipeline_specific_config)`: updates the configuration of the current pipeline with the given `pipeline_specific_config`, which is a dictionary object. This is particularly useful if a job consists of more than one pipeline, then any pipeline specific configurations can be provided using this API method.
- `run()`: starts the execution of the job. The framework will first devise an execution plan for the job and then execute the pipelines of the job according to this plan. Pipelines that do not depend on the outputs of others will be executed in parallel.

An example of a data processing job created using these API methods is shown in Listing 6.3.

3.2 Configuration files

In order to run a job, apart from creating it using ServerlessMR API methods, it is also necessary to provide its job configuration file **static-job-info.json**. This configuration file records general information of a job. Another configuration file **driver.json** which records specific properties of provisioned AWS Services, can be provided optionally. Fields of these two JSON configuration files are explained in the Appendix sections A.1 and A.2.

Chapter 4

Implementation

ServerlessMR is implemented in Python due to the language's prevalence in the data science community and its short cold start time on AWS Lambda (around 240ms [63]).

4.1 Overall Architecture

The final architecture of ServerlessMR is shown in Figure 4.1. There are five main components in this architecture: Driver, Mapper, Coordinator, Reducer and S3 Shuffling Bucket.

From a high level, the workflow of a simple ServerlessMR job (shown in Figure 4.1) that consists of a map stage followed by a reduce one is as follows:

1. The driver's responsibility is to set up the different components for a job execution: serverless functions (mapper, coordinator and reducer Lambda functions), S3 shuffling bucket and output data storage. After they are set up, it will start the job execution by invoking the mapper Lambda functions with the input keys that each mapper needs to process as parameters. These input keys can either be the item ids in a DynamoDB table or object keys in an S3 input bucket.

2. Each mapper reads the data of its corresponding input keys (specified as parameters to the Lambda function) from the input storage. For each input key, its corresponding input value (content) is fed into the map function. Once all the input values are mapped, it will write the outputs (intermediate data) into the S3 shuffling bucket. Note that the intermediate data is partitioned into different files. Its final step is invoking the coordinator Lambda function to signal the termination of a mapper.

3. A coordinator Lambda function has a mechanism to check whether all mappers have finished executing. If that is the case, then it will schedule the reduce stage by invoking the reducer Lambda functions with their corresponding reduce keys (S3 object keys of the intermediate data in the shuffling bucket) as parameters.

4. Each reducer starts off by reading the intermediate data of its corresponding reduce keys (specified as parameters to the Lambda function). Then it runs the

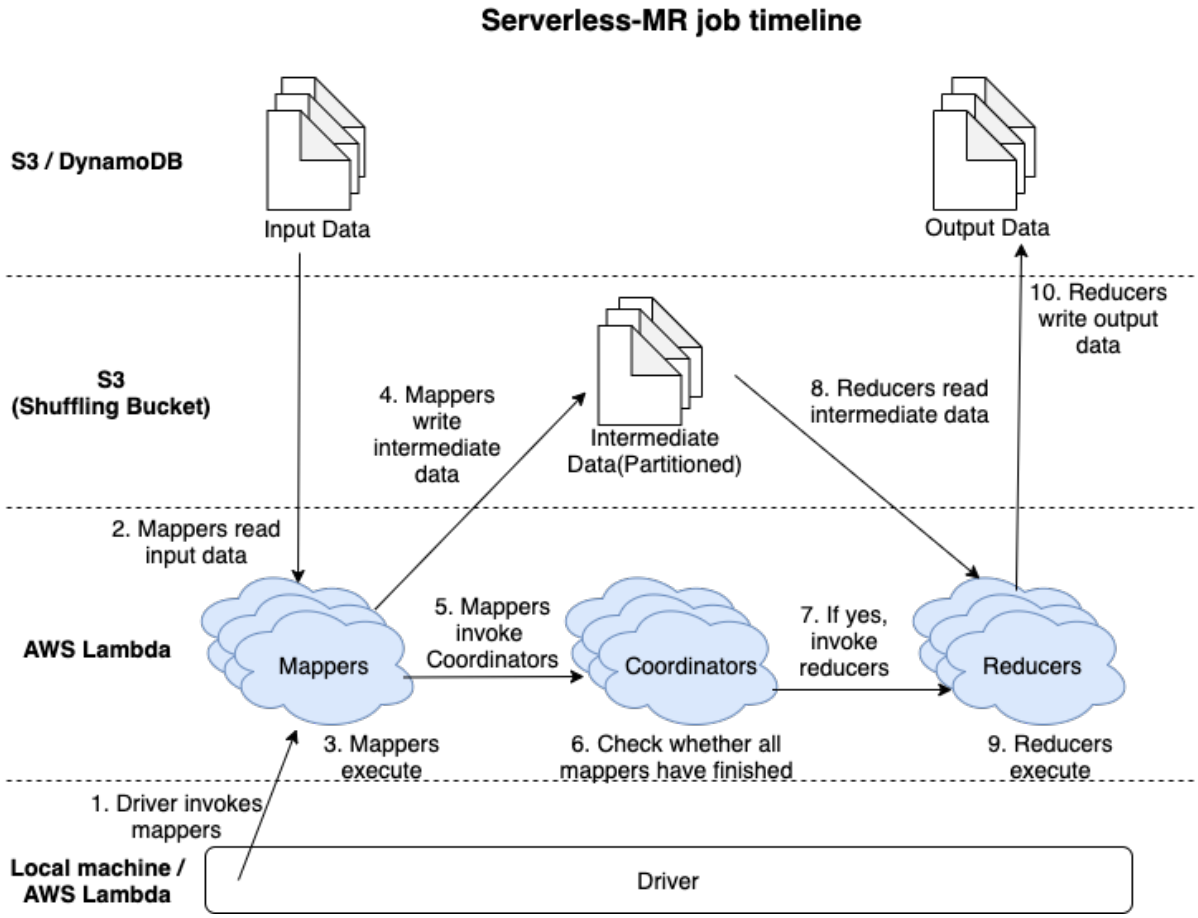


Figure 4.1: The final ServerlessMR architecture.

reduce functions on these intermediate data and the output is written into a user-specified output storage.

Design Choice

It was challenging to come up with a completely serverless MapReduce architecture that is both performant and scalable. In standard MapReduce frameworks such as Hadoop, its architecture can benefit from the fact that worker nodes are long-running and persistent which allow them to keep job states and intermediate data. However, it is very difficult to replicate the way these persistent worker nodes operate with stateless serverless functions that have execution time limits. For instance, in Hadoop, between the map and reduce phases, intermediate data can be directly transferred from a mapper node's local disk to a reducer node, omitting the need of a remote storage for intermediate data. In a serverless setup, transferring data from a mapper to a reducer is no longer possible since a mapper needs to finish executing as soon as possible or else, it might time out. Therefore, a remote storage to store intermediate data will be required. The chosen storage is AWS S3 because it is highly scalable, reliable, fast and inexpensive, and can store any kinds of data as documents.

Server-oriented MapReduce architecture can also benefit from data locality at worker nodes which speeds up reads and writes of data. Moreover, the use of resource managers such as YARN and Mesos makes the master and worker nodes fault-tolerant. These two properties can be replicated in a serverless setup using AWS services to a large extent. Data locality can be mostly achieved as a S3 bucket and a Lambda function can be configured to be in the same region, hence speeding up reads and writes. Fault-tolerance can be achieved naturally by the fact that failure in a Lambda invocation causes it to retry three times.

This framework was intended to be designed in such a way that no persistent long-running machines are required at any phases of a job execution. In other words, all computations and coordinations between different stages in a job are to be carried out by serverless functions. AWS Serverless MapReduce architecture (explained in Section 2.6.2) offers this property by having a component called the coordinator that schedules reduce stage after the termination of map stage. However, a drawback of this architecture is that reducers have to recursively perform the reduce stage on two intermediate outputs at a time until all outputs are reduced. It makes this architecture unscalable and inefficient. Shuffling will be essential to avoid this issue.

Recalling shuffling from the Background chapter, it distributes all intermediate key-value pairs such that all values associated with same intermediate key are grouped together and passed to a node that will perform the reduce phase. The aforementioned limitation of serverless functions manifests itself the most during the shuffling phase. In an ideal scenario, mappers would be available long enough to transfer intermediate data to reducers on an on-demand basis (even during the map phase). However, the execution time limit of 15 minutes makes this approach infeasible. For a serverless setup, a new shuffling method had to be employed. We decided to adopt the shuffling approach of Corral (explained in Section 2.6.2), which uses a S3 bucket as the back-end for shuffle. This approach along with the reason why it was chosen, is explained in detail in Section 4.1.3.

To get the best out of both worlds, our architecture introduces shuffling to AWS serverless MapReduce architecture, which changes the coordinator and reducer parts of the architecture completely. Earlier prototypes of ServerlessMR were built, reusing a minor part of AWS serverless MapReduce code. However since then, many advances have been made to implement more sophisticated components and features.

4.1.1 Driver

The role of the driver is to set up all the different components that will be needed in a job execution. When running a job, it performs the following series of steps:

1. It first creates the S3 shuffling bucket if it is not existent and a DynamoDB table called Stage-State which keeps track of the number of executors finished in each stage. A stage is a map or reduce phase.

2. It then registers Lambda functions of different components: mapper, reducer and coordinator. For mapper and reducer, the pointers of the user-provided map and reduce functions are pickled[1]. Both the pickle files and copies of user-provided functions' source files are stored inside the framework's working directory. Subsequently, it zips the source file of the mapper Lambda handler function along with the map function's source file, its pickle file and other different utility functions. This generated zip file is uploaded as source code to the mapper Lambda function which is created using the boto3 SDK. The same series of steps applies to the creations of reducer and coordinator Lambda functions, except that for the coordinator, there is no need to pickle any user-provided functions.

3. Input key batches are created next. Each mapper will process one batch. The driver first retrieves all the input keys and their content sizes. It then uses an algorithm to construct the batches of keys. The maximum memory size of Lambda to hold the input data (let's call this, maximum data memory) is set to be 70% of the provisioned Lambda memory size since the rest 30% is reserved for class objects. The algorithm first calculates the average size of an input key. The batch size is 1 if the maximum data memory is larger than the average size of an input key and the total number of input keys is smaller than the maximum number of concurrent Lambda invocations. A batch size of 1 means that one mapper will only process one input key. If the aforementioned conditions are not satisfied, then the batch size is set to the floor of (maximum data memory / average key size) assuming that the maximum data memory is still larger than the average key size.

4. One mapper is invoked per batch asynchronously. A batch of keys is passed to a mapper Lambda function as `event` parameters.

5. The output storage is created if it does not exist yet. Then the driver waits until the completion of the job. To check for job termination, the driver compares the current number of files in the output storage with the expected total number of output files. The expected total number of outputs is the number of reducers specified in the reduce stage, since one reducer produces one output. After that, it calculates the cost of the entire job execution. The total cost is the sum of the two constituents: data cost and Lambda cost. The former consists of S3 shuffling bucket storage cost and operations cost (GET and PUT). It is calculated by taking into account the sizes of the intermediate data files and the total number of GET and PUT operations performed on the bucket. The Lambda cost is calculated by looking at the total amount of time all Lambda functions have spent running. Information of each executor's runtime is stored as metadata of the intermediate S3 data objects in the shuffling bucket.

6. After the job finishes and the cost is calculated, the driver cleans up the job execution by removing all the Lambda functions registered and the Stage-State

[1]**Pickling**: the process whereby a Python object hierarchy is converted into a byte stream, and "unpickling" is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy [69].

table. It was decided that the shuffling bucket should be not deleted since the users might be interested in the intermediate results of the processing.

4.1.2 Mapper

A mapper is an AWS Lambda function that runs a map function on the inputs. Mapper Lambda function is invoked with the parameters: the pickle file paths of the map and partition functions, input keys to be processed and id of the mapper. A mapper first loads pointers of the map and partition functions using the functions' pickle files and source code, which are stored locally. Then, it streams the input data by reading one input key at a time and applying the user-provided map function on that key's value. This optimisation of streaming the input data prevents a mapper from having to load input values of all the given input keys at once into memory. Once all input values have been fetched and processed, it then splits the intermediate outputs into different bins by running the partition function on the key of each intermediate pair. Next, for each bin, its intermediate pairs are written to the S3 shuffling bucket with the object key `/bin-$[id]/$[mapper-id]` where `id` is the bin id and `mapper-id` is id of the mapper. If there are n bins, then a mapper will write n different S3 objects, each one to its corresponding S3 'directory'. Lastly, it invokes the coordinator Lambda to signal the termination of a mapper.

4.1.3 S3 Shuffling Bucket

This bucket stores intermediate pairs that will be later processed by reducers and also, acts as the back-end for a stateless shuffle. S3 was chosen over other storage, because of its high availability, scalability and bandwidth at a cheap price. The shuffling method used is adopted from Corral and it works as follows. Prefixes of S3 object keys are used to distinguish between files that correspond to different partition bins. In a map stage, intermediate key-value pairs are partitioned into n bins where n is the number of reducers in the subsequent stage. Pairs in each of the bins are written to one intermediate file, which is persisted to the shuffling bucket using object key `/bin-$[id]/$[mapper-id]` where `id` is the bin id (between 1 and n) and `mapper-id` is id of the mapper (between 1 and number of mappers). Then each reducer will fetch all the intermediate files in its designated bin and process them. An illustrative diagram showing how this shuffling method works is depicted in Figure 4.2.

This shuffling approach was chosen because it does not introduce any further components which might add overhead to the system. One problem with this shuffling approach is that it might end up with too many intermediate files since the number of intermediate files generated is equal to the number of mappers multiplied by the number of subsequent reducers (n). This can slightly affect the cost and performance of the framework. Because of this issue, other shuffling approaches that revolve around S3 have also been considered. One alternative is to let each bin have only one intermediate file and all mappers append their intermediate pairs that belong to this bin, to this file. This would greatly reduce the number of intermediate files generated as it

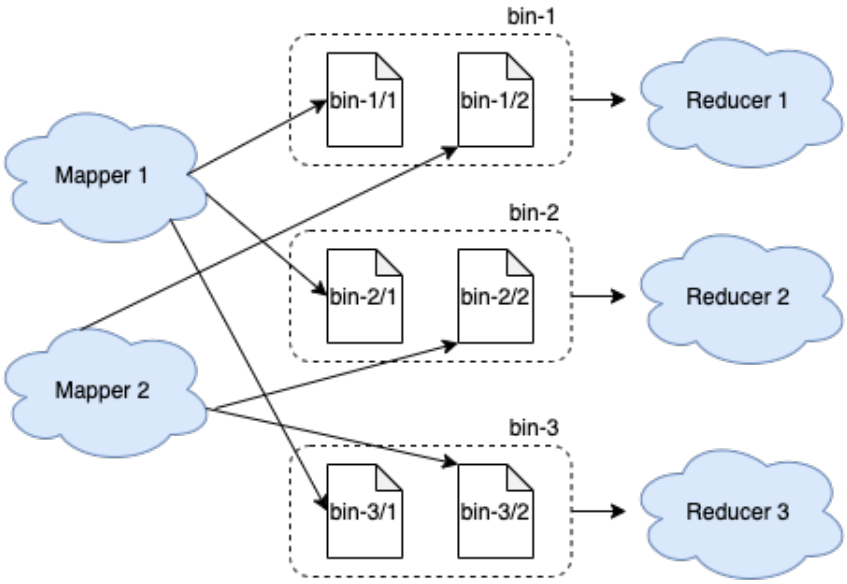


Figure 4.2: ServerlessMR shuffling technique.

limits each shuffle stage to generate only n number of intermediate files. However, this approach has a bigger issue of having race conditions. S3 does not support the append operation to an existing object/file. The behaviour of this operation can be replicated with two interactions with S3: read current object from S3 and append the data locally and then overwrite the existing object with the new one. However, because S3 does not support transactions, there would be a race condition if two mappers try to perform this append operation on the same intermediate file concurrently. If both of them read the same intermediate file and append their data separately, then only the latest write will be persisted as the other write will be overwritten and hence, lost.

Other alternatives had other issues, hence we decided to use the initial approach and trade off performance and cost against correctness. We do believe that the performance and cost of the framework will not be greatly affected by this design choice because S3 provides high-bandwidth reads and writes, and S3 storage cost is based on the amount of data stored rather than the number of objects. However, a large number of objects will lead to many reads and writes, which can slightly increase the S3 request cost.

4.1.4 Coordinator

When a mapper terminates, it invokes the coordinator Lambda function. The coordinator first checks whether all mappers have finished executing by performing an atomic increment of 1 on the Stage-State table and then comparing the newly updated count with the total number of mappers scheduled. If both numbers are equal, then the map stage has terminated and the reducers should be invoked. In that case, the coordinator will collect the S3 object keys of all intermediate files and split them into partition bins according to their object keys. Then for each bin, a reducer is invoked with the object keys in that bin as parameters. On other hand, if the two numbers are different, then

the coordinator will exit.

Race condition 1 - S3 event trigger

There were two race conditions that revolved around the coordinator. The first one is related to how the coordinator Lambda was invoked before. In the initial implementation, a mapper did not invoke the coordinator when it finishes running. Instead, an S3 PUT request (write) on the S3 'directory' of the last bin (n^{th}) would trigger the coordinator Lambda. For a job whose map function takes very small amount of time, a race condition could occur if the enablement of the S3 PUT event trigger takes longer than a mapper's execution. An event of a mapper's termination will be missed and the count in the Stage-State table would never be equal to the total number of mappers scheduled. This would lead to the reducers never being invoked. There were several ways to fix it, but a simple solution was to make mappers call the coordinator Lambda function when they finish. This solution also removes the overhead of configuring the S3 event trigger and does not require adding extra components to the system.

Race condition 2 - Map stage's termination

Initially, the Stage-State table was not present and the way the coordinator checks whether all mappers have terminated was by comparing the number of intermediate files in the last (n^{th}) bin's S3 'directory' with the total number of mappers scheduled. However, with this method, a race condition could occur in the following scenario. In a job execution, mapper A finishes earlier than mapper B (the last mapper), and while A's coordinator is checking, B has written its intermediate files to the shuffling bucket. Then using this naive approach to check for map stage termination, both coordinators will think that they were triggered by the last mapper. This will result in the reduce stage being scheduled twice. This race condition does not make the reduce stage produce incorrect results as each reducer output is overwritten by a same one. However, these redundant reduce stages will increase both the computational and storage costs.

A storage that supports transactions or a distributed coordination service such as Apache ZooKeeper was needed to solve this race condition. This ruled out storage such as S3. Upon researching online and pondering the problem, the following two prominent solutions were devised:

- **AWS SQS, AWS SNS and CloudWatch Alarm**: Instead of invoking the coordinator Lambda using S3 PUT event, an SQS queue can be set up such that the coordinator is triggered when the number of messages in the queue reaches the total number of mappers scheduled. When a mapper finishes running, it would add a message to the designated SQS queue. Since SQS cannot trigger a Lambda based on a condition, AWS SNS and CloudWatch Alarm are required. CloudWatch Alarm could be configured so that an 'AlarmAction' is triggered as soon as the SQS queue reaches a certain number of messages. This 'AlarmAction' will notify a SNS topic, which in turn triggers the coordinator Lambda. This approach will fix the race condition since the coordinator Lambda would be triggered only

once and that is when all mappers have finished execution and have posted a message to the queue [7].

- **AWS DynamoDB**: Since DynamoDB supports transactions and atomic updates, a counter could be kept in a DynamoDB table and is incremented by 1 every time a mapper finishes. Each time a coordinator is called, it performs an atomic increment of 1 on the counter. Within the same request, it also gets the updated value as the response. This new value can then be used to check against the total number of mappers scheduled. There would be no race condition in this solution since the updates are atomic, and hence only one coordinator will satisfy the condition of the counter being equal to the number of mappers scheduled. The counter could also be updated in the mapper instead, and would only invoke a coordinator if the condition is satisfied. However, this was discarded because pushing extra computation to the mappers can jeopardise the mappers from timing out.

At the end, the second approach using DynamoDB was selected because it does not require adding many different components to the framework, which might impact its performance and cost. In approach 1, triggering the coordinator will require 3 steps.

Theoretic race condition

The coordinator collects S3 object keys of the intermediate files using the `ListObjects` operation. However, since S3 offers eventual consistency for this operation[2], in theory, a possible race condition is that this operation does not return the object keys of all intermediate files to the coordinator. Then some intermediate files might be missed out. However, in practice, this race condition has not been encountered. This might be due to the short propagation time of a new S3 object. Due to its extreme rareness, there has not been a need to deal with it. Nevertheless, if in the future, this race condition occurs, then a possible fix is to use S3 `waiter`. A mapper would write all its intermediate files to S3 and then uses the `waiter` to wait until all its writes are propagated across S3. Only after that, it would invoke the coordinator. This way, it is guaranteed that the `ListObjects` operation of the coordinator would return the object keys of all the intermediate files.

4.1.5 Reducer

Similar to a mapper, a reducer receives the path of its reduce function's pickle file along with its id and list of S3 object keys to process. The reducer first loads its reduce function pointer and reads all the intermediate pairs from the specified object keys. Then, the intermediate pairs are sorted according to its key so that all values of the same key are grouped together. Afterwards, the reducer iterates through the sorted intermediate data and executes the reduce function on each unique intermediate key with its set of values. The output file of this reducer is persisted to the specified output storage. Each reducer generates one output file.

[2]If a process writes a new object to S3 and immediately lists keys in its bucket, the `ListObjects` operation might not return the new objects until the change is fully propagated.

Chapter 5

Advanced Features

5.1 Combiner Function

One of the optimisations in Google's MapReduce implementation and Hadoop is the optional use of a combiner function. There are cases where there is significant repetition on the intermediate keys produced by each mapper and the user-provided reduce function is commutative and associative, for instance, the word counting example in Section 2.3.1. As word frequencies tend to follow a Zipf distribution, each mapper will produce a large number of pairs of the form: (X, 1) for common words such as 'of'. In ServerlessMR, without a combiner function, all the counts of one word would be sent over the network to S3 and then retrieved by one reducer to produce one number for this word. To minimise the amount of data transfer and storage, ServerlessMR allows users to specify an optional combiner function that partially merges this intermediate data before it is persisted to S3. For instance, in the word counting example, using its reduce function as the combiner function, 1000 records of ('of', 1) in a mapper will be merged into 1 record of ('of', 1000). This optimisation minimises the amount of intermediate data transferred and stored, therefore reducing the storage cost and IO latency.

If a combiner function is to be used, then it is executed on each mapper. In most cases, the combiner and reduce functions are the same [51]. ServerlessMR provides both the options of using the default combiner function which is the reduce function in the subsequent reduce stage and using a custom user-provided combiner function that is different to the reduce one. Note that the output of a combiner function is written to intermediate files, which are then retrieved and processed by reducers.

In terms of implementation, with this new feature, the driver now also needs to pickle the combiner function's pointer. Moreover, the function's source and pickle files are also included in the zip file of mapper's source code. Later, when the mapper is invoked, the path of the combiner function's pickle file is passed in as a parameter. The mapper will first load the combiner function's pointer and apply its map function on the input pairs like before. Then it will run the combiner function on the intermediate data produced by the map function. The outputs of the combiner function will be

split into different bins using the partition function, then each bin will be stored in the shuffling bucket.

One of the main challenges encountered when implementing this feature was that a combiner function need not have the same output type as a map function. As explained in the Background section on MapReduce, a map function has the following signature[1]: (k1, v1) →[(k2, v2)]. A combiner function has the same function signature as a reduce function: (k2, [v2]) →(k2, [v2]). The different output types would require a reducer to handle its inputs differently depending on whether a combiner function is used. More importantly, in jobs which consist of multiple map and reduce stages chained together, having the reduce output value's type as list [v2] can cause confusion on the expected input types of map and reduce functions in later stages. Therefore, it was decided that ServerlessMR's reduce and combiner function would not follow the signatures specified in the MapReduce paper and instead would be changed to: (k2, [v2]) →[(k3, v3)] where (k3, v3) can be of the same type as (k2, v2). The inspiration for this comes from the book "Data-Intensive Text Processing with MapReduce" [12] and it was reassuring to find out that Hadoop's reduce function also follows this signature [19].

5.2 Serverless Driver

In the initial implementation, the driver must run on a stateful machine (usually, a developer's local machine). However, what if the users want to set up a cron job that runs a data processing task every 6 hours? With the initial implementation, a machine would be required to be available and to set up the task every 6 hours. This machine might be idle the rest of the time. In order to make this common data processing pattern more feasible and cheaper, an idea was to run the driver on an AWS Lambda function. This Lambda function would then be invoked when the job associated with it is to be run.

The driver's main responsibilities are to create the zip files of the different Lambda functions' source code, register the Lambda functions with the created zip files and invoke the mappers. An AWS Lambda function can perform all these operations without any problem, except the zipping part. Zipping on a Lambda function can be tricky because it does not allow writing to any of its directories apart from the `/var/tmp/` directory. There were two main design choices to be made: firstly, a choice between zipping the Lambda functions' source code locally or on Lambda and secondly, where to register the Lambda functions.

[1]Symbol () indicates the type pair. Symbol [] indicates the type list. Function signature A →B means taking input of type A and producing output of type B.

5.2.1 Design Choice: Zipping

The zipping of different Lambda functions' source code could be done on the developer's machine when the driver Lambda function is being set up. The deployed artifact of the driver Lambda function would then already contain different zip files of source code that belong to the mapper, reducer and coordinator. When the driver Lambda is invoked, it would not have to perform zipping and can directly register the Lambda functions using existing zip files. This approach's advantage is that it does not have the overhead of creating zip files every time the driver Lambda is invoked. However, this is unlikely to improve the performance of a job, since zipping small files tend to be fast. Alternatively, different Lambda functions' source code could be zipped in the driver Lambda. However, zipping on an AWS Lambda function can be trickier due to the limited directories where files can be written. At the end, as the first approach is simple without introducing any limitations to the framework, it was selected. Note that the explanation in this section only covers the case when a driver Lambda function is registered from a developer's machine. Another way to register a driver Lambda function is through ServerlessMR's Web App where a different mechanism is used. (Explanation of this mechanism is provided in the Web Application section).

5.2.2 Design Choice: Registering Lambda functions

Another decision that had to be made was on where to register the job's Lambda functions (mapper, coordinator and reducer):

- In the driver Lambda function: the advantage of this approach is that there would not be a large number of Lambda functions registered at once since after each run of a job, its Lambda functions (excluding its driver Lambda) are deleted and recreated again in a new run. Its disadvantage is the overhead of registering a job's Lambda functions in every run.
- In the local machine (during set up): this method's advantage is that there would be no overhead of registering the Lambda functions in every run. Its disadvantage is that there would be a large number of registered functions at one point in time, which makes it hard for users to look for specific functions on AWS Console. This is especially the case if users have several jobs registered.

It was decided that a job's Lambda functions would be registered in the driver Lambda since it works in the users' favours and its overhead is negligible. By inspection, registering an AWS Lambda function takes less than 1 second. Hence, it is unlikely to affect the setup time of a job hugely.

5.2.3 Implementation

To register a driver Lambda function from a local machine, firstly, any user-provided functions' pointers are pickled and zip files of different Lambda functions' source code are created. Then, the created zip files of the functions along with most of the ServerlessMR source files are zipped and the driver Lambda function is registered using the

generated zip file. This zip file also includes source code of the handler function for the driver Lambda. This handler function is called when the job associated with it, is to be executed. When it is invoked, it performs the same tasks as the driver, except that it does not create the zip files of the different Lambda functions' source code.

Note that the term 'registered job' refers to a job that has a registered driver Lambda function associated with it and hence, can be run using this serverless driver mode.

5.3 Local Execution

Often, users have to deploy a data processing job to a cloud environment in order to test if it executes correctly. This process can be costly and long as the deployments of cloud resources can take a long time. The solution to this problem is the local execution feature. It aims to simulate the execution of a job on the cloud environment locally so that users can test their code locally. This feature requires a component that can simulate AWS Lambda and other AWS services such as S3, DynamoDB. A custom implementation of such component from scratch is infeasible due to the size and complexity of each service running asynchronously, and the fact that the inner workings of the cloud systems are unknown. An off-the-shelf library that has been around for a longer time would be a better choice because it probably is the state-of-the-art local simulation of AWS services.

Several different libraries were tried. The most noteworthy ones were localstack[2] and moto[3]. Both are easy to use and can be easily incorporated to the codebase of ServerlessMR. Moreover, both cover the majority of AWS services. At the end, localstack was selected because it provides the following functionalities that moto lacks [22]:

- Error injection: localstack allows injecting errors that occur in the real cloud environment, for example, ProvisionedThroughputExceededException which is thrown by Kinesis or DynamoDB if the amount of read/write throughput is exceeded.

- Isolated processes: In moto, services are often hard-wired in memory. For instance, when a message is forwarded to a SQS queue from a SNS topic, a local hash map is used to look up the queue endpoint. On the contrary, localstack services are separate processes that communicate via HTTP. This better resembles the real cloud environment.

- Web dashboard: localstack provides a web dashboard that allows users to see local AWS services that are currently used and also, further details about each individual service. For instance, the source code of each registered Lambda function can be viewed in this dashboard.

[2] `https://github.com/localstack/localstack`
[3] `https://github.com/spulec/moto`

5.3.1 Implementation

All AWS service clients `boto3 client` have to be initialised differently if they were to communicate with localstack simulated AWS services. A local endpoint URL has to be passed in as a parameter to the initialisation. Each service has a different port number. For example, for S3, the port number is 4572. Apart from that, all `boto3 resource` and its methods had to be replaced by `boto3 client` and its equivalent methods as `localstack` does not support them [4]. A new field "localTesting" was added to static-job-info.json to denote whether a job is to be executed locally or on the real cloud environment. This flag is then used to determine the initialisation of a `boto3 client`. An example of an S3 client initialisation is shown in Listing 5.1. Note that if the illustrated piece of code is being run on a Lambda function, then the `localhost` URL is replaced by the environment variable `'LOCALSTACK_HOSTNAME'` (which is set automatically by localstack).

Listing 5.1: New initialisation of AWS boto3 client for S3.

```
if static_job_info[StaticVariables.LOCAL_TESTING_FLAG_FN]:
    if in_lambda:
        local_endpoint_url = 'http://%s:4572' %
            os.environ['LOCALSTACK_HOSTNAME']
    else:
        local_endpoint_url = 'http://localhost:4572'
    self.client = boto3.client('s3', aws_access_key_id='',
                      aws_secret_access_key='',
                      region_name=StaticVariables.DEFAULT_REGION,
                      endpoint_url=local_endpoint_url)
else:
    self.client = boto3.client('s3')
```

In order to improve the user experience of this feature, there are several steps which are automated. For instance, a new field "localTestingInputPath" is added to static-job-info.json where users can specify the local directory path of the input files and the framework will automatically upload these files to the designated input storage.

End-to-end tests can also be set up locally to assert whether the job computes the correct results and users will no longer have to compare outputs of a job with its expected results manually. Furthermore, users can even incorporate local test cases as part of the CI process of a job's deployment to the production.

[4] Both `boto3 client` and `boto3 resource` allow an user to interact with any AWS services. The difference resides in `boto3 client` providing a low-level service access and `boto3 resource` providing a more high-level and object-oriented API.

5.4 Multiple Storage Types

If S3 were the only storage type supported by ServerlessMR and a job's data is stored in a DynamoDB table, then data transfer costs would be incurred by moving input data from DynamoDB to S3 and vice versa for the output data. Hence, ServerlessMR provides support for multiple storage types. Input data can be read from different storage types and output data can be persisted to different ones. Currently, S3 and DynamoDB are supported. Support for any other storage can be added easily by creating two classes that implement the following two interfaces respectively. The interface methods to handle the input are as follows:

- `set_up_local_input_data(self, input_filepaths, static_job_info)`: this method is only called when a job is executed locally. It uploads the input data that is stored locally in the file paths `input_filepaths`, to the specified input storage.
- `get_all_input_keys(self, static_job_info)`: retrieves all input keys from the specified input storage. Input keys refer to the keys of the input pairs, but depending on the storage type, an input key can be a primary key of a DynamoDB table or a S3 object key.
- `read_value(self, input_source, input_key, static_job_info)`: given an `input_key`, reads its content from the specified storage `input_source`.

The interface methods to handle the output are as follows:

- `create_output_storage(self, static_job_info, submission_time)`: creates the output storage if it does not exist. For DynamoDB, the table name is formed using the job execution's start time (`submission_time`) and its output storage name specified in `static_job_info`. For S3, the output bucket name is the specified storage name and `submission_time` forms part of the prefix of the output objects' keys.
- `write_output(self, id, outputs, metadata, static_job_info, submission_time)`: writes the `outputs` and some `metadata` to the output storage. Metadata includes information such as runtime of an executor, maximum memory usage, etc. For S3, this information of an executor can be stored as metadata of its produced output file. However, for DynamoDB, an additional table is required to store this metadata information. In this table, `id` of the executor is the primary key and metadata is stored as JSON formatted string.
- `list_objects_for_checking_finish(self, static_job_info, submission_time)`: If the output storage is an S3 bucket, then this method returns object keys of the output files. For a DynamoDB output table, it returns ids of the completed executors, retrieved from the metadata table.
- `check_finish(self, completed_executors, final_executors, static_job_info, submission_time)`: This method checks whether a job has completed by comparing the number of `completed_executors` (obtained from the previous method) with the number of `final_executors`. If they are equal, then cost of the output storage is calculated and returned.

All methods in the output handling interface take the parameter `submission_time` (start time of a job execution). By forming the names of the output storage or files using this parameter, a job that is run multiple times will not have its previous runs' outputs overwritten by its latest execution.

The type of the input pairs that are passed in to the map function (in the initial stage) is different depending on the input storage type specified.

5.4.1 S3

If the input storage type is S3, then the input key is an S3 object key and hence, its input value is the binary content of the corresponding object. For ServerlessMR, we assume that a large dataset is partitioned into several smaller S3 objects on the input bucket. Using this assumption, it was decided that in ServerlessMR, if the input storage type is S3, a map function will operate on an entire object (document) rather than on each line of the object. This approach allows users to apply map function on a document and enables data processing jobs that require the relative position of a record. A good example is a weighted word counting job, where higher weights are assigned to words that appear earlier in the document, for instance, words in the first 25% of the lines get higher weights.

5.4.2 DynamoDB

DynamoDB allows reading a particular record or a set of records that satisfy a given condition from a table. It manages the partitioning of data for users. In other words, one single large DynamoDB table is partitioned internally and stored in different nodes [13]. Hence, the assumption made is that it is more likely for users to have the dataset stored in a single large table rather than several smaller ones. This is the reason why, in a job execution, ServerlessMR only retrieves input from one DynamoDB table. If the input storage type is DynamoDB, then the input key is the primary key of the input table. The framework supports both types of primary keys (partition key and composite key).

Because DynamoDB is a NoSQL database which can store nested objects, it was challenging to come up with a representation for the input value. This representation is important since it affects how a map function parses its input value. One representation is a string that is a concatenation of all the attributes of an item with comma. It would allow users to operate on all the attributes of an item and resemble the representation of an input value from an S3 input storage. However, it has two disadvantages:

- Less feasible for nested objects: For nested objects such as:

```
{
 "name": "Hang",
 "information":
    {"school": {
       "marks: [90, 100, 80]",
       "tests": ["Maths", "Computer Science", "Physics"]
```

```
        }
      }
    }
```

it is tricky to convert the attribute `information` into a string. Even if characters such as '{' and '}' are used to denote the hierarchy of an attribute, users would still have a hard time later parsing this string.

- Potential performance overhead of concatenating a very long list of attributes for each item (record).

Another representation is a dictionary. This solves the two problems with the first approach since a dictionary can represent well the structure of a DynamoDB attribute. A nested attribute can be represented by a nested dictionary. Apart from that, to avoid reading attributes that are not used by the map function and hence reduce the cost and read latency, a new field `inputProcessingColumnsDynamoDB` was added to static-job-info.json. Users can specify all the attributes which a map function will need, in this new field. Then, the framework will only populate the input value dictionary with the specified attributes. The attribute keys in the dictionary are the same as in the DynamoDB table.

Note that the input value is a dictionary of attributes of one item (record) rather than of the whole table. This is because in this case, the input key is the primary key which uniquely references to only one item.

5.5 Pipeline-based API

Initially, for simplicity, ServerlessMR's API was designed to be based on a decorator pattern. A user-provided map function was not passed to the framework as a function pointer, instead it was decorated with a specific decorator annotation `@map_handler`. For a reduce function, its decorator annotation is `@reduce_handler`. An example of using this old API is shown in Listing 5.2. These decorators dynamically add behaviours to an user-provided function, such that it becomes a Lambda handler function. Listing 5.3 illustrates how a decorated map function is converted into a Lambda handler function of a mapper.

Listing 5.2: Example of map and reduce functions using ServerlessMR's old API.

```
@map_handler
def map_function(outputs, input_pair):
    ...

@reduce_handler
def reduce_function(outputs, intermediate_data):
    ...
```

Listing 5.3: The decorator method map_handler.

```
def map_handler(map_function):
    def lambda_handler(event, context):
        # Read input data.
        map_function(outputs, input_pair)
        # Write the intermediate data to S3.
    return lambda_handler
```

When ServerlessMR only supported jobs which consist of a single map and reduce stage, this decorator API was adequate, even though it had some issues. For example, a map or reduce function that is decorated cannot be invoked directly since any calls to them are intercepted by their decorator methods. This makes testing of these functions difficult. Later, when ServerlessMR started to allow chaining of multiple map and reduce stages (explained in the Multi-stage section), this old API became hard to use. Listing 5.4 demonstrates how a user will define a map-map-reduce job using this old API. There are two main problems with it. Firstly, since the decorator methods need to be hard-coded manually up to a certain number of stages, this approach would not work for jobs that require many stages. The number of stages allowed would be limited by the number of decorator methods defined. Secondly, it can be hard for users to visualise a data pipeline that is defined using this API since often, different stage's map and reduce functions would be defined in different files. There is not a centralised place where the execution order of the stages is defined.

Listing 5.4: Chaining of map and reduce functions using decorator approach.

```
@map_handler_1
def map_function_1(outputs, input_pair):
    ...

@map_handler_2
def map_function_2(outputs, input_pair):
    ...

@reduce_handler_3
def reduce_function_3(outputs, intermediate_data):
    ...
```

A new API was needed to provide a better user experience and enable the implementation of more advanced features such as support for multi-stage MapReduce jobs. Looking at Hadoop and Spark which are the two most popular data processing tools, users specify map and reduce functions through function pointers and objects. This option is popular among other data processing frameworks as well such as PyWren and Corral. It provides flexibility in terms of function naming and where they reside. The initial design of this new API is illustrated in Listing 5.5 which shows the definition of a map-map-reduce job. Each method of the API has an additional parameter (the last one) that indicates the stage number of the provided function. This would solve the first issue with the old API as it can scale to any number of stages. However,

the second problem would still persist to a certain extent. Even though compared to the old API, this new approach would allow users to easier visualise a data pipeline as the job would be defined in a centralised place, numbering the stages can still be a confusing way to help users have an overview of a job. Apart from that, users could accidentally configure a provided function with the wrong stage id since essentially the framework is pushing the responsibility of assigning stage ids to functions, to the users.

Listing 5.5: The definition of a map-map-reduce job using function pointers.

```
from serverless_mr.main import ServerlessMR

from user_job.map import extract_data_1
from user_job.map import extract_data_2
from user_job.reduce import aggregate_revenues
from user_job.partition import partition
from user_job.combine import combiner_function

serverless_mr = ServerlessMR()
serverless_mr.set_map_function(extract_data, 1)
serverless_mr.set_map_function(extract_data_2, 2)
serverless_mr.set_combiner_function(combiner_function 2)
serverless_mr.set_partition_function(partition, 2)
serverless_mr.set_reduce_function(aggregate_revenues, 4, 3)

serverless_mr.run_job()
```

A modified version of the previous approach is a pipeline-based API. This aims to illustrate the execution order of the stages and their corresponding functions with the order in which they are specified. This would better solve the second issue with the old API. Listing 5.6 shows an example of a map-map-reduce job using this API. At the end, it was decided that this pipeline-based API would be implemented as it provides a much better user experience than the old API and helps users better visualise a long data pipeline as you would see in the Multi-Stage and Multi-Pipeline sections.

Listing 5.6: The definition of a map-map-reduce job using pipeline-based API.

```
from serverless_mr.main import ServerlessMR

from user_job.map import extract_data_1
from user_job.map import extract_data_2
from user_job.reduce import aggregate_revenues
from user_job.partition import partition
from user_job.combine import combiner_function

serverless_mr = ServerlessMR()
serverless_mr.map(extract_data_1).map(extract_data_2)
    .combine(combiner_function).shuffle(partition)
    .reduce(aggregate_revenues, 4)
```

```
serverless_mr.run()
```

5.5.1 Implementation

A library that can serialise and deserialise the pointer of an user-provided function, was required. Upon trying several different serialisation libraries, the top 3 candidates were `pickle`, `cloudpickle` and `dill`. All three provide similar functionalities and meet our needs. However, due to the ease-of-use of `pickle` and the fact that it is the default Python serialiser (which means that there would be no need to install it on Lambda), the library `pickle` was chosen. The series of steps involved in 'delivering' an user-provided function to its corresponding Lambda handler function is listed below:

- When an user-provided function's pointer is passed in as a parameter to an API method, the framework first finds the path of its source file. If a function pointer is serialised in one place and deserialised in another, then it is crucial to replicate the original directory structure of its source file (where it is serialised) in the place where it is deserialised. This is because if a function is imported using a relative path X and this pointer obtained through X is serialised, then later when the pickled function pointer is to be deserialised, the relative path X must still remain valid. For instance, in Listing 5.6, since `extract_data_1` is imported from the relative path `user_job.map` and serialised, later in order for a mapper to successfully deserialise it, the relative path `user_job.map` must still remain present. In other words, the function must still reside in the same relative path. For this reason, the directory structure of the source files of a provided function pointer is replicated in the working directory of the framework. This is achieved by cloning the directories where the function is defined and copying its source files into the framework directory.

- Once all functions' source files and its directories have been successfully replicated in the framework directory, the driver pickles (serialises) each of the function pointers and generates a pickle file per pointer. Each pickle file is named in the format: `[function-type]-[stage-id].pkl` where `function-type` is the type of the function (e.g. a map function) and `stage-id` is the id of the stage that it belongs to. Later, when an executor Lambda function is created, the driver includes both its corresponding function's pickle file and source files in the zip of the Lambda function's source code.

- When an executor Lambda function is invoked, the path of its corresponding function's pickle file is passed in as a parameter. Before the support for multi-stage MapReduce jobs were provided (next section), the driver could make the assumption that the mappers it invokes are always in stage 1 and the coordinator could assume that the reducers it invokes are always in stage 2. Therefore, the paths of the pickle files could be inferred and constructed easily.

Note that combiner and partition functions are handled in the same way as map and reduce functions.

5.6 Multi-stage

There are many scenarios where users would like to create a sequence of MapReduce jobs to completely transform and process the data. This is better than putting everything in a single MapReduce job whose only map and reduce functions will be very complex. This feature is particularly important for ServerlessMR due to its serverless nature. Since Lambda functions can only execute for a maximum of 15 minutes, it is preferred to have a job with many short-stages rather than a job with a few long-stages. A stage is a map or reduce phase that is applied to the data. A long and complex stage might risk its Lambda executors from timing out. Apart from that, having a sequence of MapReduce jobs allows much complex data pipelines to be defined such as a map-map-reduce-map-reduce job.

Initially, it was hard to figure out what kinds of multi-stage processing patterns should be supported. Upon researching, the following patterns seemed to be well-received in the data science community:

1. `Map+ Reduce Map*`: a job can have 1 or more map stages, followed by a reduce stage and 0 or more map stages. Hadoop supports this pattern through its classes `ChainMapper` and `ChainReducer`.

2. `(Map Reduce)*`: chaining of `(Map Reduce)` is a popular pattern. This enables execution of iteration algorithms, for instance, applying an algorithm (same map and reduce functions) to the data for a fixed number of iterations. Note that this pattern also enables non-iterative algorithms that have a long sequence of `(Map Reduce)` where the map and reduce functions in each part of the sequence are different.

   Interestingly, even without this multi-stage feature, this pattern can already be achieved by breaking down a job that consists of a sequence of `(Map Reduce)` into individual MapReduce jobs that are executed one after another. Output of one job is then piped into the input of the subsequent one. Although this approach works, it is tedious since users will be required to define one configuration file per `(Map Reduce)`. Moreover, for every `(Map Reduce)`, they need to make sure that its output storage is the same as the input storage of the next one.

3. Multi-pipeline: A pipeline in ServerlessMR is defined as a sequence of map and reduce stages that are executed one after another, with output of one stage being the input of the next one. A new pattern is combining outputs from different pipelines and sending them into a new pipeline as input. Each of the independent pipelines can execute concurrently. This pattern is described in more detail in the section of Multi-pipeline.

In order to support the first and second patterns, 3 out of 4 types of 'joint' between stages are required. Table 5.1 shows the types of 'joint' and whether they are supported by ServerlessMR. The joint type reduce→reduce is not supported, because in most use cases, it does not make sense to have such joint. A reduce stage aims to

aggregate a set of values belonging to a key. Having one reduce stage after another means further aggregating values of a key that have already been aggregated by the first reduce stage. Often, this does not make much sense since the second aggregation could have been performed on the first reduce stage. However, there is an exception in which this kind of joint might be needed. If the output key of the first reduce stage is different to its input key, then a second reduce stage would make sense since it acts on a different key domain compared to the first reduce stage, and shuffling between the two stages would be required. Considering that the only use case of reduce→reduce is rare and that is why it is not supported by Hadoop, it was decided that it would not be implemented in ServerlessMR. Nevertheless, users will still be able to write jobs that require a reduce stage following another by using reduce→map →reduce where the map stage applies an identity function to its input.

Type of joint	Supported	Shuffling Required
map→map	YES	NO
map→reduce	YES	YES
reduce→map	YES	NO
reduce→reduce	NO	YES

Table 5.1: Types of joint supported.

An example of a pipeline using the 3 supported joint types is shown in Listing 5.7.

Listing 5.7: Example of a pipeline using all the supported joint types.

```
serverless_mr = ServerlessMR()
serverless_mr.map(map_function_1).map(map_function_2)
    .reduce(reduce_function_1, number_of_reducers).map(map_function_3)
    .reduce(reduce_function_2, number_of_reducers)
```

5.6.1 Design Choice

This feature introduced two changes to the system. Firstly, a pipeline could have more than 2 stages. Secondly, 2 more joint types (map→map and reduce→map) should be allowed apart from map→reduce.

The architecture of the framework remained the same. There were two methods to accommodate the first change:

- **One coordinator per joint**: For each joint between two stages, there will be one coordinator Lambda function which exclusively handles the transition between those two stages. Each coordinator will only hold transition information that is specific to its assigned joint. This approach is simple as the coordinator will behave in the same way as before, except that it will need to handle more joint types. However, it has the disadvantage of having to create (s - 1) number of

coordinator Lambda functions where s is the number of stages. Hence, at one point in time, there might be a large number of Lambda functions registered.

- **One coordinator for all joints**: For an entire pipeline, there will only be one coordinator Lambda function that handles the transitions of all joints. This method is more complicated than the first one because there has to be a mechanism for the coordinator to determine for which two stages it is currently doing the transition. Apart from that, this coordinator will hold transition information of all the joints. This approach has the advantage of only having to register one coordinator Lambda function per job, no matter how many stages the job has.

The second change required recording more information about each transition. Initially, information of each joint (transition) was stored, however, this did not work for the multi-pipeline pattern and hence, it was decided that information about each stage would be stored instead. The exact underlying reason for this adjustment is discussed in more detail in the next section. For each stage, the following information is stored:

- Name of the executor Lambda function in this stage.
- Number of executors in this stage.
- Path of the function's pickle file: the function is either a map or reduce function.
- Path of the combiner function's pickle file: this is optional - it is only indicated in a map stage that is followed by a reduce stage.
- Path of the partition function's pickle file: this is optional - it is only indicated in a map stage that is followed by a reduce stage.
- Stage type: type 1 refers to stages that need to perform shuffling (such as the map stage in map→reduce) whereas type 2 refers to stages where shuffling is not required (map→map and reduce→map).

To handle a transition, the coordinator will need to look up stage information of the two stages involved. The above information of all the stages is stored in the file `stage-transition-info.json`.

5.6.2 Implementation

With this new feature, it is possible now to have a map stage followed by another map stage, in which case, shuffling will not be required. For this reason, in the codebase, there are 3 types of stages: `map`, `mapShuffle` and `reduce`. A map stage that is followed by a reduce, is a `mapShuffle` since apart from applying a map function to the input, it also needs to perform the shuffling.

Provided a multi-stage job such as the one in Listing 5.7, the framework will first copy the provided functions' source files into the framework directory. It will also construct a list of `stage` objects. This list is sorted by their execution order. There are 3 types of `stage` objects that correspond to the 3 different types of stages. Each `stage` object

contains information that is specific to its stage property. However, all three share the following information: (map or reduce) function pointers and relative paths of its function's source files (in the framework directory). A `mapShuffle stage` object will also records its partition and combiner functions' pointers and relative paths of their source files. Then this list is passed to the driver.

When creating the executor Lambda functions, the driver will iterate through this list of `stage` objects. For each `stage` object, it will first pickle its function pointer. For `mapShuffle stage` object, its partition and combiner functions will also be pickled. Then a zip file that comprises the corresponding Lambda handler of the current stage, its function's source files and pickle files, is created. This stage's executor Lambda function is registered next with the created zip file. An environment variable in the Lambda function is set to the id of the current stage. Afterwards, driver adds the current stage's information to the dictionary `stage-transition-info` whose key is stage id and value is a dictionary that contains the fields listed in the previous subsection. Once the iteration is over, the driver persists `stage-transition-info` to a JSON file and the coordinator Lambda function is registered with a zip file that includes this JSON file. The driver will then proceed as before.

The executors behave in the same way as before, except that they now have to check whether they are in the two special stages: first and last. If an executor is in stage 1, then it will retrieve the contents of the input keys from the specified input storage, otherwise input will be read from the shuffling bucket instead. If an executor is in the last stage, then it will write its outputs to the specified output storage, otherwise outputs will be persisted to the shuffling bucket instead.

When an executor that is not in the last stage finishes running, it will call the coordinator Lambda function. The stage id of the executor will be passed in as a parameter. The coordinator uses this stage id to retrieve the corresponding stage information from `stage-transition-info.json`. The stage id will also be used to get the number of executors completed at this stage from the Stage-State table. Then it will proceed as before: checking whether the number of executors completed is the same as the number of executors scheduled. If that is the case, then the coordinator will schedule the next stage using its stage information.

5.7 Multi-pipeline

A pipeline in ServerlessMR is defined as a sequence of map and reduce stages that are executed one after another, with output of one stage being the input of the next one. The third pattern that is introduced in the section of Multi-Stage is multi-pipeline. This pattern allows combining outputs from several different pipelines and sending them into a new pipeline as input. Independent pipelines of stages can be executed in parallel. An example use case is if a user wants to apply some transformations to data in an S3 storage, apply different transformations to data in a DynamoDB table and then aggregate the outputs from the two. Listing 5.8 shows how one could define such a

job using ServerlessMR. Three pipelines have been defined. Pipelines 1 and 2 process the data from S3 and DynamoDB respectively and pipeline 3 aggregates the outputs from the two. This essentially forms a DAG[5] of pipelines where later pipelines depend on the earlier ones.

The example job could have been defined by having three separate jobs that correspond to the three pipelines. The first two jobs would be configured to output to the same storage and the third one will read its input from that storage. However, the advantage that the multi-pipeline feature offers is its parallelism. Pipelines that do not depend on others can execute in parallel and independently. This is similar to the idea of DAG of stages in Spark. The example job's DAG of pipelines is shown in Figure 5.1. Pipeline 3 depends on pipelines 1 and 2, hence it cannot start its execution until both of them have completed. However, pipelines 1 and 2 can execute in parallel. Note that shuffling cannot be performed between stages in different pipelines. Hence, the first stage of a dependent pipeline can never be a reduce.

Listing 5.8: An example of a multi-pipeline job.

```
serverless_mr = ServerlessMR()
pipeline1 = serverless_mr.config(config_pipeline_1)
    .map(extract_data_s3).reduce(reduce_function_1, 4).finish()

pipeline2 = serverless_mr.config(config_pipeline_2)
    .map(extract_data_dynamo_db).reduce(reduce_function_2, 2).finish()

pipeline3 = serverless_mr.config(config_pipeline_3)
    .merge([pipeline1, pipeline2]).map(map_function_3)
    .reduce(reduce_function_3, 5).run()
```

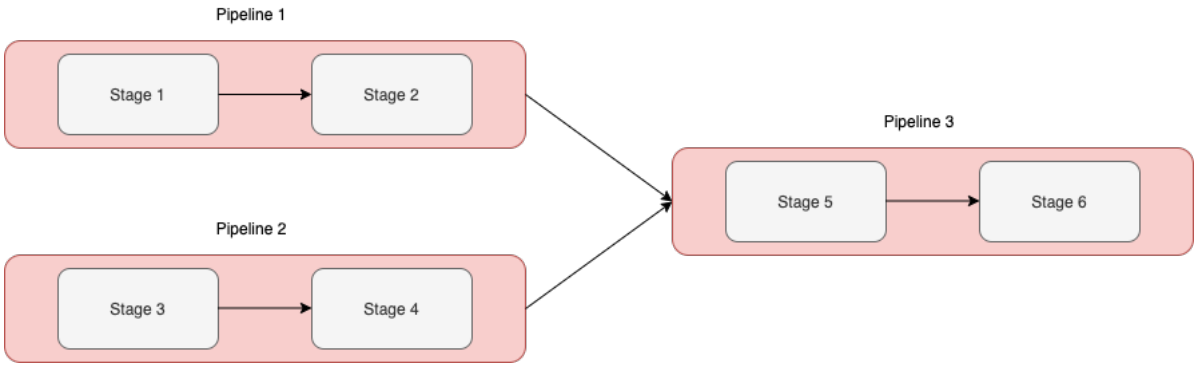


Figure 5.1: Pipelines DAG of the example job.

[5]DAG: Direct Acyclic Graph

5.7.1 Design Choice

For a multi-pipeline job, intra-pipeline stages can be executed in the same way as before. A key challenge with this feature is coming up with a method to handle transitions between stages from two different pipelines. There are two main approaches:

- **Each pipeline as a job**: In this approach, each pipeline runs as an individual job. Each one will have its own shuffling bucket, job configuration file and coordinator Lambda function. A new type of coordinator that handles transitions between stages from two different pipelines will be needed. It will hold information about dependencies between pipelines. An executor in the last stage of a pipeline will call this new type of coordinator before it finishes.

  An advantage of this approach is that users can express jobs that have pipelines DAG of any shape. Since each pipeline has its own output storage, its another advantage is that users will be able to persist outputs of intermediate pipelines to their specified output storages. This is particularly useful if for example, users want to persist a specific pipeline's outputs to a DynamoDB table. However, this approach has a disadvantage. It will require one S3 shuffling bucket per pipeline. Since an AWS account is limited to have 100 S3 buckets at once, the number of pipelines that a job can have will be limited and even, the number of multi-pipeline jobs registered at once will also be constrained. Interestingly, this will not affect the cost since S3 cost does not depend on the number of buckets. One bucket with n files costs the same as 10 buckets with the n files distributed among them.

- **All pipelines as a job**: Another approach is running all the pipelines as one single job, which only has one shuffling bucket and one coordinator Lambda function. Different to the first approach, there will not be a new type of coordinator to handle transitions between stages from two different pipelines. Instead, this responsibility is pushed to the standard coordinator that handles transitions between intra-pipeline stages. It will hold information about pipelines dependencies in order to determine whether the next stage of a transition belongs to the current pipeline or a different one. An advantage of running the coordinator on Lambda is its unlimited scalability. In this approach, this prevents the coordinator from becoming a bottleneck in the system.

  As with approach 1, an advantage with this method is that users can define jobs that have pipelines DAG of any form. Apart from that, since it assigns one shuffling bucket per job, the number of pipelines a job can have is not limited and the number of multi-pipeline jobs registered at once is much less constrained compared to approach 1. However, its disadvantage is that outputs of intermediate pipelines cannot be written to a specific storage other than the job's designated S3 shuffling bucket.

Both approaches would offer the same API, except that approach 1 would require users to write one different configuration file per pipeline. For instance, in Listing 5.8, if

approach 1 were used, `config_pipeline_1`, `config_pipeline_2` and `config_pipeline_3` would have to be completely different since each pipeline is a different job. For approach 2, the three pipelines would share most of the configuration fields. These common configurations would be populated in the file static-job-info.json. Only configurations that are specific to a pipeline are provided using the method `config()`. Hence approach 2 offers a better user experience.

Apart from that, approach 1's disadvantage is more severe than approach 2's one as it limits the number of pipelines a job can have. With regard to the second approach's drawback, if the outputs of certain intermediate pipelines have to be persisted to a separate data storage, this can be achieved by either breaking up the job into two or move the data from the shuffling bucket to the designated storage after the job's termination. Because of the aforementioned reasons, approach 2 was selected.

For transitions between intra-pipeline stages, a DynamoDB table (Stage-State) is used to determine whether a stage has completed and to avoid race conditions between different coordinator invocations. In transitions between inter-pipeline stages, there has to be a similar mechanism in a distributed setting that helps a coordinator invocation decide whether a dependent pipeline could be started. Upon researching, a Leetcode algorithm [21] could be adapted for our distributed setting and be suitable for our use case. Its pseudocode is described below, note that in our case, a node is the representation of a pipeline and a directed edge from A to B indicates that pipeline B is dependent on A:

1. Initialise a queue, Q to keep track of all the nodes in the graph with 0 in-degree. In-degree of a node is the number of arcs directed towards that node.

2. Iterate over all the edges in the input and create an adjacency list and also a map of node to its in-degree.

3. Add all the nodes with 0 in-degree to Q.

4. The following steps are to be done until Q becomes empty:

   (a) Pop a node from Q. Let's call this node, N.

   (b) For all the neighbours of this node, N, reduce their in-degree by 1. If any of the nodes' in-degree reaches 0, add it to Q.

   (c) Process the node N.

   (d) Continue from step 4.a.

As the coordinator runs on Lambda, a remote storage is necessary to store the live in-degree of each pipeline. Updates to this information must be atomic, otherwise race conditions would occur. Hence, DynamoDB is used because of its support for atomic updates and transactions.

5.7.2 Implementation

Provided a multi-pipeline job, the framework will first construct a list of `pipeline` objects. A `pipeline` object holds specific configuration of a pipeline, its list of `stage` objects and ids of the pipelines that it depends on. As before, the source files of each stage's functions will be copied into the framework directory.

This list of `pipeline` objects is then passed to the driver. In the step of creating executor Lambda functions, the driver initialises a list `invoke_pipelines`, to keep track of all the pipelines that do not depend on others, i.e., pipelines with an in-degree of 0. For each `pipeline` object P, the current job's configuration is overwritten by P's specific configuration fields. The driver then checks whether P has no dependencies on others. If that is the case, then it adds P to `invoke_pipelines` along with the necessary information for its invocation. It later loops through P's list of `stage` objects and sets up each stage's executor using the right configuration (details of how these executors are set up are given in the Multi-stage section). Apart from that, during the iteration of the `pipeline` objects, the following information is collected:

- An adjacency list which records dependencies between pipelines is created and implemented as a dictionary. Each entry in this dictionary consists of a pipeline id (as the key) and a list of ids of its dependent pipelines (as the value). For instance, if in a job, pipelines 3 and 4 depend on 1, then its adjacency list will contain the entry `1: [3, 4]`.
- In-degree of each pipeline.
- Mappings of stage id to its corresponding pipeline id.
- The ids of the first and last stages of each pipeline.

All this information, except the in-degree of each pipeline is then persisted locally as JSON files. When the driver creates the coordinator Lambda function, these JSON files will be included in the zip of its source code. This information will be useful for the coordinator when handling transitions of inter-pipeline stages. The in-degree information is used to initialise the DynamoDB table In-degree that keeps track of each pipeline's live in-degree value.

After all the Lambda functions are created, the driver initiates the execution of the job by starting the first stage of each pipeline in `invoke_pipelines`. Note that all the executors are invoked asynchronously by the driver, hence each pipeline runs in parallel.

An executor behaves in the same way as before and is not aware of the pipeline that it belongs to. When an executor finishes running, it calls the coordinator Lambda function with its stage id as the parameter. The only change in the coordinator's behaviour is how it schedules the next stage. If the next stage is in the same pipeline as the current one, then it is scheduled in the same way as described in the Multi-stage section. However, if the next stage belongs to a different pipeline, then the coordinator first gets the pipeline id of current stage, P. A list of pipeline ids that are dependent on P

are retrieved from the adjacency list. For each dependent pipeline DP from the list, its in-degree value in the In-degree table is atomically decremented by 1 and the updated value is returned in the same request. If this in-degree value is equal to 0, then the coordinator will schedule the first stage of DP since all the pipelines that it depends on have finished. Note that the input keys for DP's first stage are the outputs from the last stages of all pipelines that it depends on.

The reason why that Leetcode algorithm was selected is because it could be adapted for our distributed setting such that there will not be race conditions between different coordinator invocations. Even if there are several coordinator invocations running concurrently and all of them are decrementing the in-degree value of a pipeline P. Since the decrements are atomic and the updated value is returned in the same request, there will only be one coordinator invocation that gets to decrement P's in-degree value to 0 and schedule P.

5.7.3 Problem: Stage transition information

Initially, the information necessary to handle each transition was stored per joint/-transition rather than per stage. This information was indexed using the joint id (the source stage id of a transition). For each transition, the following information was stored (a transition is from a source stage to a destination stage):

- Name of the executor Lambda function in the destination stage.
- Number of executors in the source stage.
- Number of executors in the destination stage.
- Transition type: type 1 refers to transitions where shuffling is not needed (map→map and reduce→map) whereas type 2 refers to transitions where shuffling is required (map→reduce).
- Path of the function's pickle file in the destination stage: the function is either a map or reduce function.
- Path of the combiner function's pickle path in the destination stage: this is optional - it is only indicated if joint type is map→reduce.
- Path of the partition function's pickle path in the destination stage: this is optional - it is only indicated if joint type is map→reduce.

This way of storing transition information made sense for multi-stage tasks since for each stage, there would always be at most one next stage. Therefore, there is one-to-one mapping from a piece of stage transition information to a transition. However, once the multi-pipeline feature was introduced, this mapping could be one-to-many since there could be multiple next stages from different pipelines. This occurs when multiple pipelines depend on one such as the example shown in Figure 5.2. Stage transition information indexed using joint id could not reference multiple transitions.

For instance, in the illustrated example, it was unclear how stage transition information indexed using joint id 2 can reference both the transitions from stage 2 to 3 and 5. For this reason, it was decided that information about each stage would be stored instead and a transition would use the stage information of the two stages involved.

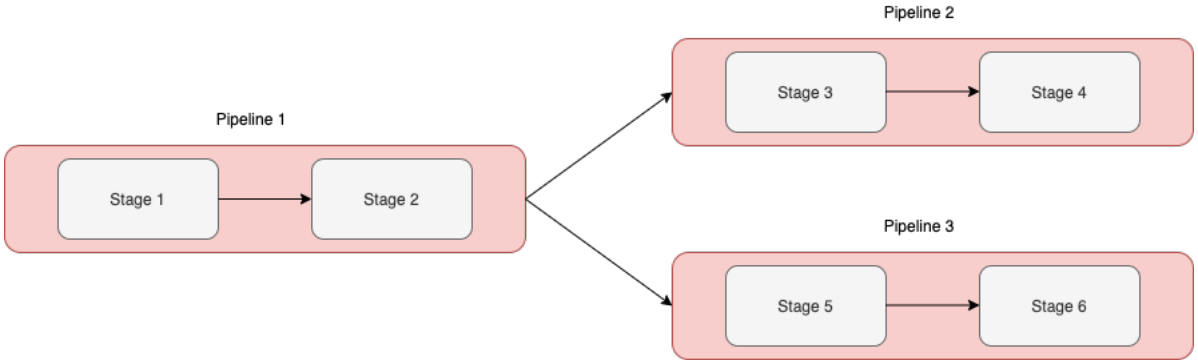


Figure 5.2: Example of multiple pipelines (2 and 3) depending on one pipeline (1).

5.7.4 Problem: Serverless Driver

The discussed implementation of this new feature works when the driver is running on a local machine. However, in serverless driver mode where the driver runs on a Lambda function, the current implementation will work with a caveat. Recall that the zip file of a coordinator's source code is created on a local machine when the serverless driver Lambda is being registered. In this mode, the stage information would be computed on the local machine and included in the coordinator's zip file. However, stage information contains fields which are data dependent such as the number of executors to schedule at each stage. Hence, the caveat is that the number of executors at each stage would be determined based on the data present in the specified input storage at the time of the job registration. This would present a problem for jobs which are executed repeatedly every X amount of time since mostly likely, the data is different every time the jobs are run. Hence, the number of executors to schedule at each stage should also change in each run. However, that would not be the case with our current implementation.

Essentially, there is a dilemma: stage information file based on the latest data cannot be created due to the difficulty in zipping on Lambda and it should not be created on a local machine when the driver Lambda is being registered because that fixes the number of executors based on the data at that time. The solution was when executing a job in serverless driver mode, the stage information will be collected on the serverless driver Lambda and then, this information will be persisted to the shuffling bucket. The coordinator will be able to access this information based on the latest data, from the S3 bucket instead of local directories.

5.8 Web Application

ServerlessMR provides the option for users to spin up a web application that makes the interaction with the framework much easier. On the web app, users can perform the following actions:

- View all the jobs that belong to the AWS account of an user in real time: these include jobs that are completed, registered and currently running. Moreover, further information about each job is available such as submission time of a job, its output storage, etc.
- Monitor any job execution's progress in real time: this is much easier than using AWS Console and tracing the logs of Lambda functions.
- Run a registered job: users can start the execution of a registered job from the web app.
- Schedule a registered job: a registered job can be scheduled to run at a specific time of day/week/month or every few minutes/hours/days. Users schedule a job by providing rate and cron expressions for frequencies of up to once per minute.
- Register a job: the web app provides a simple "text editor" page where users can write a data processing job from scratch and register it.
- Modify a registered job: the source code of any currently registered job can be viewed. Additionally, that source code can be modified and the job can be re-registered with the changes. Apart from that, using this option, a new job can also be registered based on another one.

Since the main focus of the web app is its features, it was decided that a front-end template would be adopted rather than building the web UI from scratch. ReactJS was the technology used for the front-end. It was preferred over others due to its high rendering performance and good development experience. The template used was from [50] and had to be adapted for our use case. The back-end consisted of a RESTful API built in Python using `Flask`. Note that this web app can also run locally, allowing users to perform the same aforementioned actions on locally-executed jobs.

5.8.1 Architecture

This web application runs on an AWS Lambda function. This brings the benefits that users do not have to worry about tearing down the web application every time they stop using it and only get charged for the time that it is used. The web app's architecture is shown in Figure 5.3. It consists of two main components: web server (on AWS Lambda) and API Gateway. The web server is both a front-end and back-end server. When an user clicks on the web application URL, the following series of steps takes place:

1. A front-end request is sent to the API Gateway, which in turn directs the request to the web server Lambda function.

2. The web server receives the request and internally routes it to the corresponding RESTful API endpoint.
3. The first request is always a front-end request and the server sends back the front-end html document and javascript files as the response. Subsequent requests are back-end requests which usually require interaction with either the S3 bucket serverless-mapreduce-job-information, DynamoDB tables Stage-State or In-Degree.
4. The response goes through the API Gateway and reaches the user.

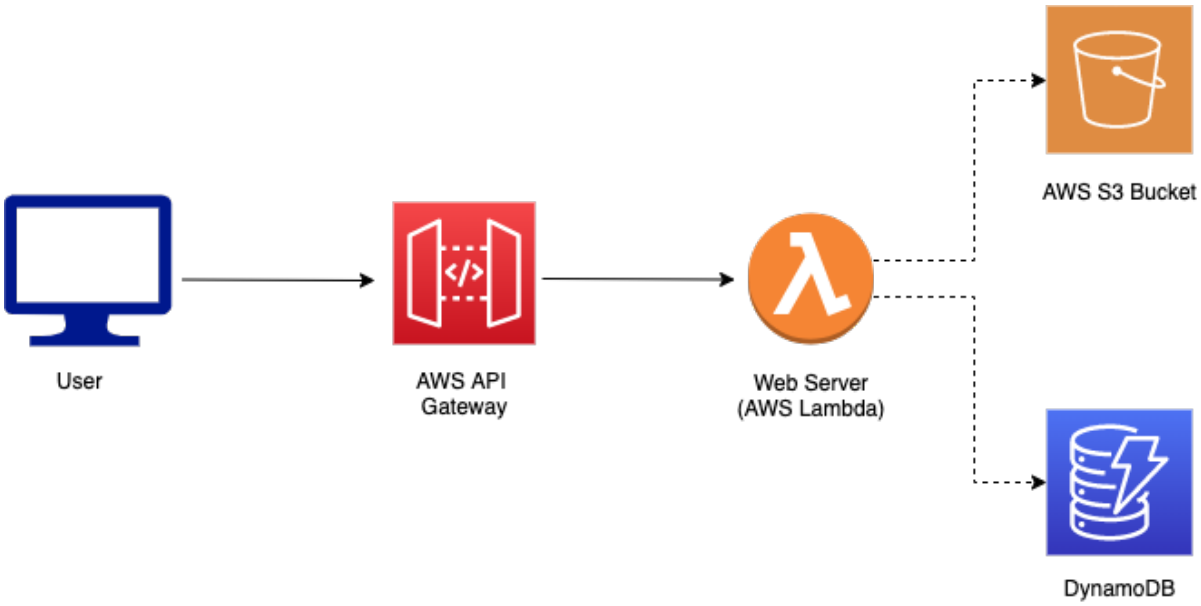


Figure 5.3: The architecture of the web application.

All the endpoints of the web server are handled by the one and only Lambda function. This function handles the path routing internally and then processes the request. This is called the monolithic pattern. Another approach is the microservices pattern which involves creating one Lambda function per endpoint and letting the API Gateway perform the path routing. The monolithic pattern has the following advantages [53]:

- Fewer cold starts since the Lambda function would be invoked more frequently and stays warm.
- Fewer Lambda functions to manage and faster deployments as there are fewer functions to deploy.

However its disadvantages are:

- Harder to debug and analyse CloudWatch logs since the function handles all types of requests.
- Updating a function might cause regression and break some other functionality.

From the user's perspective, the drawbacks of monolithic pattern are inapplicable since users will not modify the web application codebase. This approach's advantages will improve the user experience of the web app. Hence monolithic pattern was chosen.

5.8.2 General Information

The web application shows information such as current active (running), registered and completed jobs (UI shown in Figure 5.4). Apart from that, for each job, it displays in-degree values of the job's pipelines, its number of completed executors at each stage and the DAG of its pipelines (UI illustrated in Figure 5.5). In-degrees and numbers of executors completed are stored in the DynamoDB tables In-Degree and Stage-State respectively. The other information is stored in a bucket called serverless-mapreduce-job-information. This bucket stores information of all the jobs that an user owns and of all the job executions that have been performed.

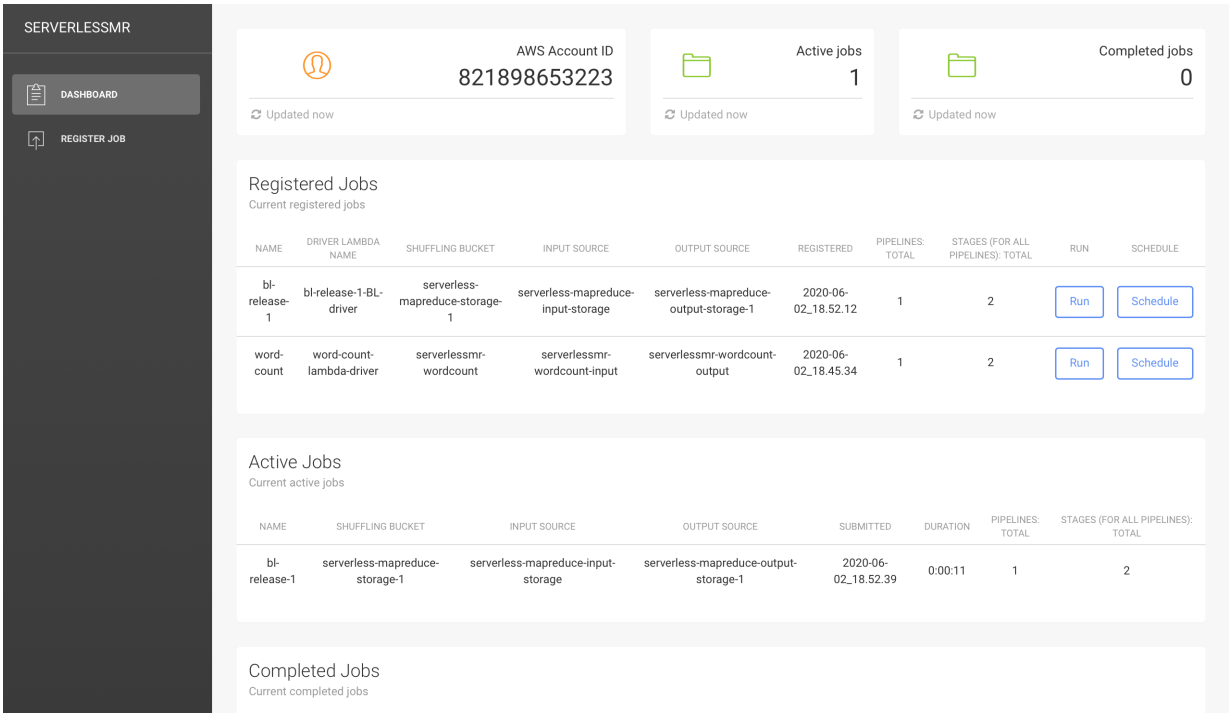


Figure 5.4: The dashboard page of the web application.

5.8.3 Challenge: Running Flask web server on Lambda

Running a flask web server on Lambda presents several challenges since flask-defined endpoints are not valid on a Lambda handler function. This would mean that all the endpoints defined using flask have to be converted into normal Python methods and path routing rules have to be explicitly defined, directing a request to its corresponding method. Apart from that, different flask utility methods such as `render_template()` and `send_from_directory()` might have to be replaced by similar-behaviour methods that are importable and usable in Lambda. For this reason, we decided to use a framework

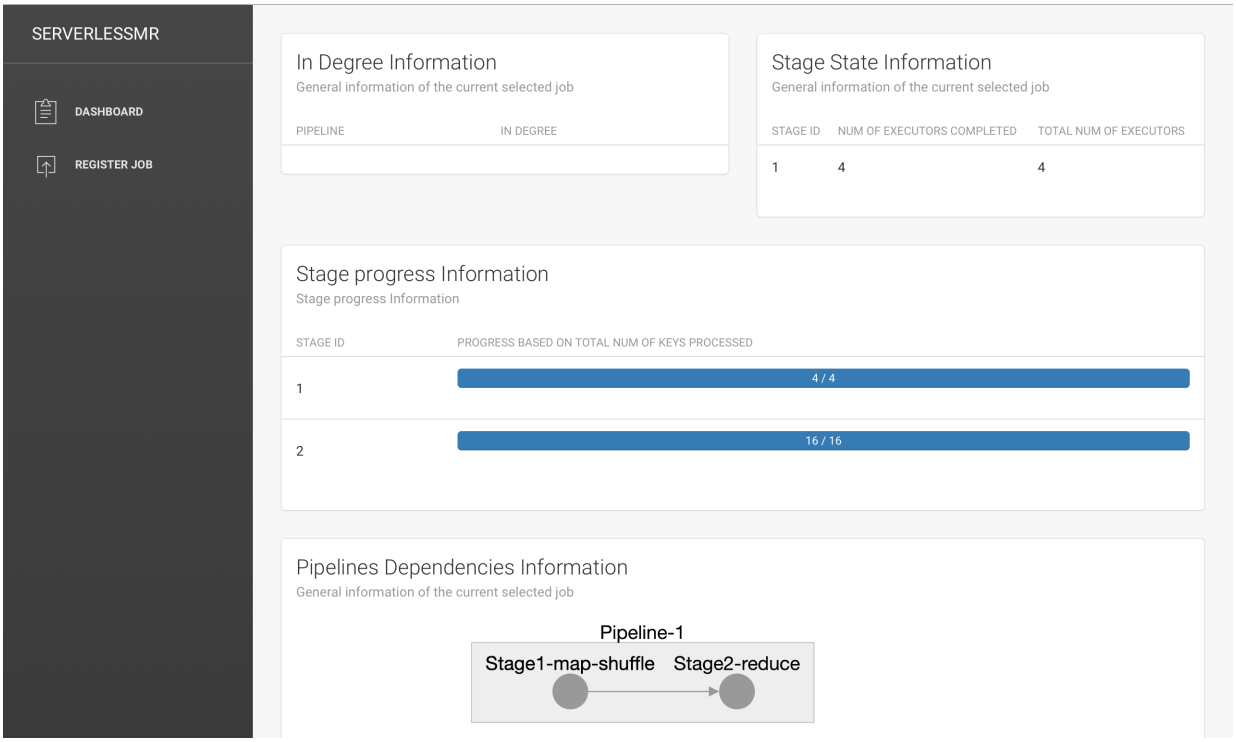


Figure 5.5: The job information page of the web application.

to deploy a serverless web application rather than deploying and configuring an API Gateway and the web server Lambda function ourselves.

The main frameworks to deploy a serverless web application are: Serverless and Zappa. Interestingly, AWS also offers a microframework called AWS Chalice that helps one to build a serverless web application that runs on Lambda. However, this solution was discarded since the back-end was already written using flask. Switching from flask to Chalice would not have introduced any benefits and would require a more extensive amount of modifications than using Serverless or Zappa. For my use case, both Serverless and Zappa are great solutions, however, the former is chosen at the end because it covers more cloud providers. The integration of Serverless into ServerlessMR involved creating a generalised `serverless.yml` file where the configuration for web app is provided such as specifying the path routing rules.

5.8.4 Feature: Job progress

Monitoring the progress of a job is important since it tells the users whether a job is making progress or stuck. Progress information of different granularity levels can be provided to the users. However, the more fine-grained it is, more information would have to be stored in a remote storage. It was hard to find the best trade-off between a granularity level and its necessary storage. The finest granularity level that can be provided is giving progress information of each executor at each stage. Both S3 and DynamoDB are candidates for the underlying technology to store information required

for such granularity. ElasticCache (AWS in-memory cache) was discarded because it was too expensive for our use case.

Even using either S3 or DynamoDB, the cost of storing such fine-grained level of progress information could be significant if a job has many executors. For instance, if in a job, there are 10000 executors in a total of 10 stages, S3 would need to have 10000 files (to avoid race conditions), each one holding progress information of 1 executor. If each executor performs 100 updates to its file, then there would be a total of 1 million S3 write and read requests (also read requests because a S3 update involves a read and write). This would incur a cost of over $5. The cost of using DynamoDB is somewhere similar. Apart from that, a job's execution time might be affected if the updates are performed too frequently. In the case of using a DynamoDB table, it might throttle due to too many concurrent update requests.

The aim was to choose a level of granularity that would not affect the framework's performance and cost, but at the same time, provides sufficient progress information to the users. At the end, due to the negative impact of keeping progress information per executor, it was decided that the overall progress of each stage will be shown to the users instead. This progress information will be based on the total number of input files processed so far by all the executors at a stage. To avoid race conditions of different executors performing updates to the same place, a DynamoDB table is used to store this information.

In cluster-based MapReduce frameworks, the presence of stragglers is common. Stragglers are executor nodes that take an unusually long time to complete one of the last few map or reduce tasks in the computation. This is usually due to reasons such as the node is being used by other processes or is faulty. Identifying these stragglers are important in cluster-based MapReduce frameworks since they limit the speed at which jobs are completed. If stragglers are discovered in a job execution, then in subsequent executions, they can be given less amount of workload and hence speeding up the overall execution of a job. In ServerlessMR, it is not necessary to identify stragglers since in each execution, the underlying machines for Lambda will be different. Moreover, Lambda with high memory setting (1.5GB) tends to have consistent performance [49]. However, it can be useful for users to identify executors that take significantly longer than others. Usually, the presence of such executors suggests that the workload is not well-balanced among executors. This could be due to a biased user-provided partition function. In the current implementation, support for identifying such executors is not provided as the framework does not show progress information of each executor. As a future extension, in a job run, the runtime times of each executor can be recorded and displayed to the users. This will help them identify executors that had more workload than others, track down the causative issues and fix them for future runs of the job.

Implementation

A new DynamoDB table called Stage-Progress is created to store progress information of each stage, by the driver. The primary key of this table is the stage id and its two

attributes are current number of input file keys processed and total number of input file keys to be processed. The second attribute of a stage is set by either the driver or coordinator before invoking that stage. This is because only the invoking component knows the total number of files that need to be processed. Input file keys can refer to the keys of S3 objects or DynamoDB record ids depending on the input storage. Using the input file keys rather than the key-value pairs passed into map/reduce functions (for instance, ('word', 1) in the word counting example), would not introduce performance overhead to the framework. If key-value pairs of map/reduce functions were used, then the driver or coordinator would have to retrieve the contents of all input files and count the number of key-value pairs in them, which is a very expensive task.

For reducers, counting the current number of intermediate files processed was infeasible. Since reducers first reads all the intermediate pairs from the specified files and then, process these pairs, there is no way to know whether an intermediate file is fully processed unless we keep track of which intermediate pairs belong to which file. For this reason, an approximation of the current number of files processed is made in reducers. This estimated number N is based on the average number of pairs in an intermediate file P. N is incremented by 1 if the number of pairs processed exceeds P.

An executor will increment the current number of processed input keys of its stage in the Stage-Progress table. Increment updates are atomic. Each executor performs the updates at a randomised time interval of 10 seconds to 30 seconds. This randomness is to avoid executors of the same stage performing the updates at around the same time. This prevents throttling in DynamoDB which could be caused by too many update requests to the same stage (record) at once. According to AWS docs [37], DynamoDB provides performance in the order of single-digit millisecond. In ServerlessMR, since there can be at most 1000 executors at one stage (due to the limit of 1000 on the number of concurrent Lambda functions), having the interval between 10 and 30 seconds means that an executor performs an update every 20 seconds on average. Therefore the time allocated for each update is around 20s / 1000 = 20ms, which is longer than a DynamoDB write latency. This is under the assumption that records with different stage ids are stored in different partitions of the DynamoDB table.

Apart from that, a new endpoint was added to the back-end API and the ReactJS component Progress Bar is used to display the progress of each stage. Note that on the front-end, progress information is polled every 5 seconds.

5.8.5 Feature: Run a job

This feature allows users to invoke a registered job from the web application without having to set up a project and pip install ServerlessMR. Any registered jobs will be displayed in the front-end and with only one click away, users can run them on any device as long as it can access the web application.

In terms of implementation, parameters used to configure the Lambda functions of a

registered job such as memory limit are stored in the bucket serverless-mapreduce-job-information. A new endpoint that takes job name and the Lambda name of the job's serverless driver as parameters, is added to the RESTful API. When this endpoint is called, the Lambda configuration parameters of the job are retrieved and used to initialise the Lambda `boto3 client`. Using this client, the given serverless driver Lambda function is invoked.

5.8.6 Feature: Schedule a job

It is common to run a data processing job at periodic time intervals or at a specific time of a day. For instance, if a job needs to be executed at 4am in the morning, a user should not wake up and invoke it. In order to better support these kinds of use cases, this feature was added to ServerlessMR. It allows users to provide a cron or rate expression[6] on the web app and a registered job will be automatically triggered at times when the provided expression is satisfied.

The implementation involved creating a new endpoint in the back-end API. When it is called, a CloudWatch event rule is created using the given cron/rate expression. Then, the provided serverless driver of the job is added as a target to the created rule. The final step is granting the permission of invoking the provided serverless driver Lambda function to the rule.

5.8.7 Feature: Register a job

Users can write a data processing job on the web app and register it. The web app provides a page that contains ReactJS forms where users can write code (illustrated in Figure 5.6). With this feature, users are able to write and register any kinds of jobs on the web app without having to set up a Python project and pip install ServerlessMR and its dependencies.

Registering a job can take 10 to 20 seconds. Since our web server runs on Lambda, it is automatically scaled if its workload increases. Hence, it was decided that the web server Lambda function will perform the registering of a job instead of delegating it to another component.

Recall the difficulty of creating zip file of a component's source code in a Lambda function. In this case, if ServerlessMR code resides in the source code directory (`/var/task/`) of the server Lambda function, then zip files will be created using (`/var/task/`) as the root directory. Since Lambda does not allow writing to any of its directories other than `/var/tmp/`, any user-provided files would be stored there. If user-provided files are now included in the zip, the paths of these files in the zip would start with `../tmp/` and would lead to many modules/files not found exception later. For this reason, the registration of a job is performed on the `/var/tmp/` directory. ServerlessMR code will

[6] All expressions listed in the AWS page (`https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/ScheduledEvents.html`) are supported.

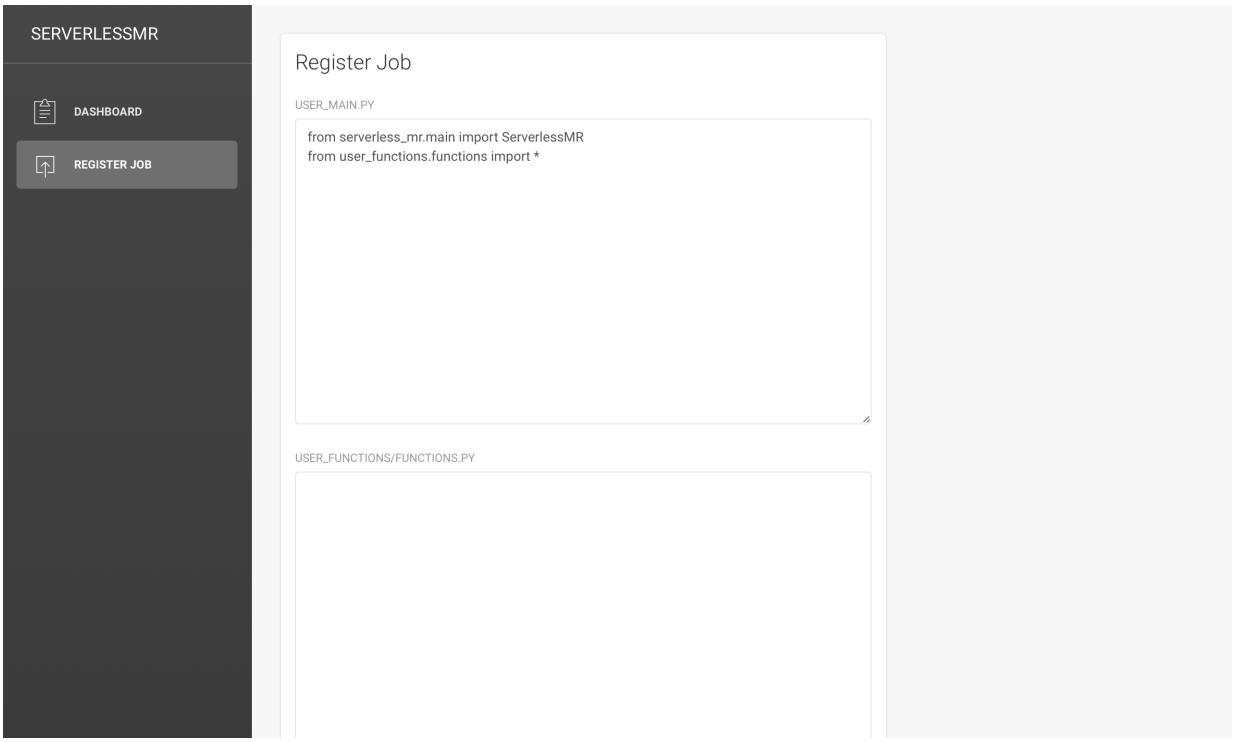


Figure 5.6: The register job page of the web application.

be in `/var/tmp/` and zipping will be performed using this directory as root, so that the zip file created will be valid and the paths of user-provided files in the zip would not begin with `../tmp/`.

For simplicity reasons in the front-end, only four files are allowed to be submitted for registering a job from the web app. These four files are the following:

- `configuration/static-job-info.json`: records general information of a job.
- `configuration/driver.json`: records properties of provisioned AWS Services.
- `user_functions/functions.py`: different map, reduce, combine and partition functions are defined in this file.
- `user_main.py`: contains the code for setting up a job to run in ServerlessMR.

Note that users can still create any kinds of jobs with these four files. The only limitation that this design choice brings is having to specify all the map and reduce functions in `user_functions/functions.py`. As a future extension, the web app could be extended such that any number of source files can be provided.

Implementation

In the front-end, a new page that contains the aforementioned four ReactJS forms and a button for registering is created. When the button is clicked, the front-end sends a POST request with the contents of the four forms as URL parameters, to a new endpoint

in the back-end. Upon receiving the request, the web server firstly downloads a slightly adjusted version of ServerlessMR code from a public S3 bucket (serverless-mapreduce-code) to the `/var/tmp/` directory. Then, content of each POST request parameter is persisted to a file with path `/var/tmp/[path]` where `path` is the corresponding hard-coded path for that parameter. For instance, the parameter `static-job-info.json`'s corresponding file path is `configuration/static-job-info.json`. Afterwards, user-provided file `user_main.py` is executed from the command line using the same environment variables as the web server. Note that in order to execute ServerlessMR's code in a non-default Python execution directory such as `/var/tmp/`, paths of different Python packages and ServerlessMR modules had to be added to the `sys.path` of the execution. This way, the other ServerlessMR modules can be imported and also, they can access standard Python packages and modules. Once the registration is completed, the web server will return a status code to the front-end.

5.8.8 Feature: Modify a job

Often, users need to make small changes to already registered jobs such as configuration changes. For instance, if users want to modify the input or output storage of a job, then having to delete the job, find the Python project where it is defined and creating a new job with the changes can be tedious and time-consuming. To make their lives easier, ServerlessMR allows them to make these modifications from the web app. On the web app, users can view the source code of any registered job, make changes to its source/configuration files and re-register the job. (The web page UI of this feature looks the same as shown in Figure 5.6, but would display different source files.)

Implementation

Firstly, when registering a job both locally and from the web app, the source files of all user-provided functions, configuration files and the main file where the job is defined are stored in the S3 bucket serverless-mapreduce-job-information. Apart from that, for each file, its original relative path with respect to the main file's directory along with where it is stored in the bucket, are recorded in an S3 object called registered_job_source_info.json.

Two new endpoints are added to the back-end API: `/get-source-files` and `/modify-job`. The first one is to retrieve all the source files of the given job. In this endpoint, the web server first reads the file registered_job_source_info.json of the specified job. Then, all source files of the job are retrieved from their corresponding S3 object keys (recorded in registered_job_source_info.json). Finally, a JSON object that contains the contents of all source files along with their original relative paths is constructed and returned as a response. The second endpoint `/modify-job` is very similar to the endpoint of registering a job. Like before, ServerlessMR's code is downloaded to `/var/tmp`. However, instead of taking four fixed URL parameters, the new endpoint can take any number of them. Each parameter is persisted to `/var/tmp/[path]` where `path` is the parameter's key. Then the provided main file of the job is run from the command line and the

status code of the operation is returned.

A new page is added to the web app. When an user clicks on a registered job, this new page makes a request to the endpoint `/get-source-files` and displays each of its source files in a ReactJS form. After changes are made to these forms, users can click on a button to submit the changes. This will send a request to the endpoint `/modify-job` with these new source files as parameters.

5.9 Testing

It can be challenging to thoroughly test a distributed data framework such as ServerlessMR, however an extensive amount of testing has been done to ensure its correctness. Below are the different ways ServerlessMR has been tested:

- **Manual end-to-end testing**: Different jobs are executed on datasets that contain added bogus records. Outputs of the jobs produced from these bogus records are compared with their expected results. Apart from that, in order to ensure result consistency (no race conditions), each job is executed several times and output files from different runs are compared using their MD5 hashes.
- **Locally executed end-to-end tests**: End-to-end test cases have been written and executed locally. In these test cases, samples of a job's outputs are checked against their expected values. Sometimes, these samples are results from added bogus records, but other times, they are from the actual data.
- **Comparison with Hadoop**: All the 11 tasks mentioned in the Evaluation chapter have been executed both in ServerlessMR and Hadoop. Sampled outputs from executions of the two frameworks have been compared. For all 11 tasks, there has never been a mismatch between the samples compared.

Chapter 6

Building an application using ServerlessMR

This chapter demonstrates how to write and run a data processing job using ServerlessMR. The aim of this is to show the general workflow when building a ServerlessMR data processing application.

6.1 Job: Word Count

The job will be a word counter that given a piece of text, outputs the frequencies of each word in the text. The steps required to set up such a job are the following:

1. Set up a Python project and pip install ServerlessMR.
2. Write the map and reduce functions anywhere inside your project directory. Listings 6.1 and 6.2 show the map and reduce functions for this job respectively.

Listing 6.1: Map function.

```
import re

def produce_counts(outputs, input_pair):
    try:
        _, input_value = input_pair
        # Split input_value (a document) into lines
        lines = input_value.split("\n")[:-1]

        for line in lines:
            # For each line, split it into words
            words = re.split("; |, |\*|\n| |:|\.", line)
            for word in words:
                # For each word occurrence, add a tuple (word, 1) to
                    outputs
                outputs.append(tuple((word, 1)))
    except Exception as e:
```

```
        print("type error: " + str(e))
```

Listing 6.2: Reduce function.

```
def aggregate_counts(outputs, intermediate_data):
    key, values = intermediate_data

    sum_counts = 0
    try:
        # Aggregate the counts of each word
        for value in values:
            sum_counts += value

        # Append (word, total counts) to outputs
        outputs.append(tuple((key, sum_counts)))
    except Exception as e:
        print("type error: " + str(e))
```

A map function returns nothing and always has two parameters `outputs` and `input_pair`. The parameter `outputs` is of type [(k2, v2)][1] where k2 is the type of the intermediate key and and v2 is the type of the intermediate value. The other parameter `input_pair` is of type (k1, v1) where k1 is the type of the input key and v1 is the type of the input value. A reduce function returns nothing and always has two parameters `outputs` and `intermediate_data`. The parameter `outputs` is of type [(k3, v3)] where k3 is the type of the output key and and v3 is the type of the output value. The other parameter `intermediate_data` is of type (k2, [v2]).

3. Define the job and submit it to ServerlessMR. In our example, a main Python script has been created and contains the code shown in Listing 6.3. Running this script will start the execution of the job. Note that in this case where no partition function is provided, the framework will use a default one.

Listing 6.3: Main file.

```
from serverless_mr.main import ServerlessMR

from word_count_map import produce_counts
from word_count_reduce import aggregate_counts

serverless_mr = ServerlessMR()
# First, apply the map function produce_counts and then the reduce
    function aggregate_counts (with 4 reducers).
serverless_mr.map(produce_counts).reduce(aggregate_counts, 4).run()
```

4. Finally, provide the configurations of the job. In your project directory, create a Python module named `configuration`. Inside this module, create the configuration file: `static-job-info.json`. Listing 6.4 shows the specified fields in this file.

[1] Symbols [] and () represent list and pair types respectively.

Note that another configuration file `driver.json` can be added to this module if you want to specify configuration parameters for AWS resources such as memory limit of Lambda.

Listing 6.4: Job configuration file static-job-info.json.

```
{
    "jobName": "word-count",
    "shufflingBucket": "serverlessmr-wordcount",
    "inputSourceType": "s3",
    "inputSource": "serverlessmr-wordcount-input",
    "outputSourceType": "s3",
    "outputSource": "serverlessmr-wordcount-output",
    "useCombine": true,
    "localTesting": false,
    "serverlessDriver": false
}
```

After the job is defined, below are the steps required to execute it on the cloud:

1. AWS credentials need to be set. This can be done by storing your credentials under the file `credentials` in ~/.aws/ directory.[2]

2. An AWS IAM role has to be created based on the file `policy.json` (stored in ServerlessMR's directory) and the environment variable `serverless_mapreduce_role` has to be set to the name of the created role. This can be done by running the script `setup.sh`, provided by ServerlessMR.

3. Finally, to start the job's execution, run the main Python file.

Note that we can also choose to register this job and run the driver on a Lambda function by setting the flag `serverlessDriver` in `static-job-info.json` to `true`. Apart from that, a web application can also be set up and used to monitor the progress of the job's execution (shown in Figure 5.5). For more details on installing the framework, creating a job, running it and the signatures of different functions, you can follow: `https://github.com/hanglili/Serverless-MapReduce-Test`.

6.2 Local Testing

Different kinds of testing can be performed. Firstly, unit tests can be written to ensure the correctness of the map and reduce functions. An example unit test for the map function is illustrated in Listing 6.5.

[2]For more details on how to set AWS credentials: `https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/setup-credentials.html`

Listing 6.5: Map function unit test.

```
class Test(TestCase):
    def test_word_count_map_method(self):
        input_pair = (1, "ServerlessMR is a serverless MapReduce framework\n")
        outputs = []

        produce_counts(outputs, input_pair)

        assert len(outputs) == 6, "test failed"
        assert outputs[0] == ('ServerlessMR', 1), "test failed"
```

Secondly, the job can be executed locally on a subset of the real data and get its results verified by the users. This will require spinning up Docker and start the `localstack` container (The flag `localTesting` in `static-job-info.json` will have to be set to `true` and the field `localTestingInputPath` will have to be added to denote the relative paths where the input files are stored).

Locally executed end-to-end test cases can also be written and executed with the `localstack` container running. The verification of the results can be automated. Listing A.1 in Appendix shows how to retrieve the outputs from the output bucket in our example and then assert whether they are equivalent to the expected results.

Chapter 7

Evaluation

7.1 Performance and Cost - ServerlessMR

In this section, ServerlessMR is tested in absolute terms using different kinds of tasks with a variety set of configuration parameter values. Evaluating it using different parameter values would allow us to discover effects of the different parameters on its performance and cost. The parameters that will be varied are: number of reducers in each reduce stage and number of mappers in the first stage.

In order to thoroughly assess ServerlessMR, a combination of tasks from the Berkeley Big Data Benchmark [34] was used. This benchmark suit was picked because it is highly-recognised and provides tasks of different kinds with the data sets. All the data sets are stored in a public S3 bucket `s3://big-data-benchmark/` as csv files. This facilitates the set up of the evaluation as it will not be necessary to generate the data manually on a machine and then transfer it to a remote storage. The data consist of one collection of HTML documents with two additional data sets that model log files of HTTP server traffic. The last two are composed by values that are derived from the HTML documents as well as some randomly generated attributes [66]. The schemas of these data sets are shown in Listing 7.1.

Listing 7.1: The schemas of the three data sets used for evaluation.

```
TABLE Documents (contains unstructured HTML documents)

TABLE Rankings (lists websites and their page rank)
pageURL VARCHAR(300)
pageRank INT
avgDuration INT

TABLE Uservisits (stores server logs for each web page)
sourceIP VARCHAR(116)
destURL VARCHAR(100)
visitDate DATE
adRevenue FLOAT
```

```
userAgent VARCHAR(256)
countryCode CHAR(3)
languageCode CHAR(6)
searchWord VARCHAR(32)
duration INT
```

ServerlessMR is evaluated against the following set of tasks [66], each one aiming to test its different aspects:

- **TASK 1 - Selection query**: a lightweight filter to find pageURLs in the Rankings data set with a pageRank above an user-defined threshold (10). In ServerlessMR, this task is only composed of a map stage.
- **TASK 2 - Aggregation query**: calculates the total adRevenue generated for each sourceIP in the Uservisits data set, grouped by the seven-character prefix of sourceIP field. This task measures ServerlessMR's performance on jobs that require shuffling. In ServerlessMR, this task is composed of the stages: map→reduce where → is read as 'is followed by'.
- **TASK 3 - Join query**: is made up of two sub-tasks that operate on two data sets. The first part consists of finding the sourceIP that generated the most revenue within a given date range. The second part requires calculating the average pageRank of all the pages visited during this date interval. This task measures ServerlessMR's performance on multi-stage jobs and join operations (between two data sets). The SQL of this query is shown in Listing 7.2. In ServerlessMR, this task is composed of the stages: map→reduce→map→reduce→map→reduce.

Listing 7.2: The SQL of the join query.

```
SELECT INTO Temp sourceIP, AVG(pageRank) as avgPageRank,
    SUM(adRevenue) as totalRevenue
FROM Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL
    AND UV.visitDate BEFORE Date(2000-01-01)
GROUP BY UV.sourceIP;

SELECT sourceIP, totalRevenue, avgPageRank
FROM Temp
ORDER BY totalRevenue DESC LIMIT 1;
```

Since the tasks provided by the Berkeley Big Data Benchmark were later shown to be short-running jobs, they were turned into longer-running tasks by adding `time.sleep(X)` to their map and reduce functions. From here on, the aforementioned tasks 1, 2 and 3 will be addressed as short-running tasks. Even though `time.sleep(X)` is not a CPU intensive operation, it does serve the purpose of prolonging the stages of a task as each input record takes more time to be processed, i.e., throughput of each executor is decreased. Apart from that, it provides the advantage of being able to precisely control the execution times of a task and each of its stages. Below is the list of long-running tasks, which were derived from the short-running ones using `time.sleep(X)`:

- **TASK 4 - Long selection query**: `time.sleep(0.00005)` is added to the processing of each input line in the map function. This task is only composed of a map stage.
- **TASK 5 - Long aggregation query**: `time.sleep(0.00005)` is added to the processing of each input line in the map function and `time.sleep(0.01)` is added to the processing of each reduce key in the reduce function. This task is composed of the stages: map→reduce.
- **TASK 6 - Long join query**: The following `time.sleep()` operations have been added to the processing of each input file/reduce key in each stage respectively: `time.sleep(30)`, `time.sleep(0.0005)`, `time.sleep(40)`, `time.sleep(0.0005)`, `time.sleep(30)` and `time.sleep(0.0005)`. This task is composed of the stages: map→reduce→map →reduce→map→reduce.

7.1.1 Setup

The setup of this evaluation involved several steps. Firstly, as with all Big Data frameworks that do not support query languages, the six tasks that are expressed as queries had to be rewritten using map and reduce functions[1]. Secondly, different metrics had to be recorded in order to estimate a task's cost and performance. These metrics had to be stored in a per-task S3 bucket since there was not a stateful central machine where they can be sent to. S3 was chosen over other databases since it is scalable and provides a write latency between 50ms to 100ms. This is acceptable in this evaluation since the latency introduced to a task is linear to its number of stages. For instance, a complicated 10-stages task will have its execution time increased by at most 1s because of this overhead. Each executor and coordinator Lambda invocations write their total execution, IO and CPU times to this per-task metrics bucket. After the task has terminated, the local machine will calculate the execution cost and performance. The common configurations used by each task are listed below:

- The driver is run on the local machine.
- Combiner functions are not used.
- Metrics required for the web app are not collected. (The field `optimisation` is set to true in the configuration file)
- Inputs are read from S3 and outputs are persisted to S3.

7.1.2 Results

Each task has been tested using two data sets of different sizes: one small and one large, shown in Table 7.1. Two experiments were conducted. The first experiment involved varying the number of mappers in the initial stage for every task and each of

[1]Source code of all tasks used in this evaluation are on: `https://github.com/hanglili/Serverless-MapReduce/tree/master/src/python/performance_functions` and `https://github.com/hanglili/Serverless-MapReduce/tree/master/src/python/performance_functions_cpu`

Tasks	Small Dataset	Large Dataset
1, 4	1node/rankings (1.28GB)	5nodes/rankings (6.38GB)
2, 5	1node/uservisits[subset of 40 files] (5GB)	1node/uservisits (25.4GB)
3, 6	1node/uservisits[subset of 40 files] (5GB) + 1node/rankings (1.28GB)	1node/uservisits (25.4GB) + 1node/rankings (1.28GB)

Table 7.1: Datasets of each task (expressed with their S3 paths followed by their sizes).

its data sets. Results of each task are shown in a table. This 'varying mappers' table contains the following columns: data set used, number of input keys allocated to each initial mapper, number of resulting initial mappers, total execution time (excluding the setup and teardown times), each stage's execution time (outputted in stage id order), total execution cost, total Lambda cost and total S3 shuffling bucket cost.

In the second experiment, for every task, the number of reducers in each stage is varied. For this experiment, the number of input keys allocated to each initial mapper is kept at 1 and only the large data set of each task is used. The results are illustrated in a table which has similar columns as experiment 1: data set used, number of reducers at every reduce stage (outputted in stage id order), total execution time (excluding the setup and teardown times), each stage's execution time (outputted in stage id order), sum of all coordinator invocations' execution times, total execution cost, total Lambda cost and total S3 shuffling bucket cost. For both tables, all the values have been obtained by averaging the results of 3/4 runs.

Varying number of initial mappers

Data Set	Num of keys/mapper	Num of mappers	Total execution time(s)	Stage execution times(s)	Total cost($)	Lambda cost($)	S3 cost($)
1node(1.28GB)	1	50	3.666	2.098	0.00293	0.00262	0
1node(1.28GB)	5	10	10.907	8.662	0.00227	0.00217	0
1node(1.28GB)	10	5	18.171	16.745	0.00215	0.00209	0
5nodes(6.38GB)	1	100	6.832	4.635	0.0124	0.0116	0
5nodes(6.38GB)	5	20	21.437	20.153	0.0105	0.0101	0

Table 7.2: Task 1's results.

Data Set	Num of keys/mapper	Num of mappers	Total execution time(s)	Stage execution times(s)	Total cost($)	Lambda cost($)	S3 cost($)
1node[:40](5GB)	1	40	24.664	8.470, 12.271	0.0140	0.0116	0.00220
1node[:40](5GB)	5	8	55.378	37.314, 11.790	0.0110	0.0104	0.000469
1node(25.4GB)	1	202	42.227	12.879, 15.739	0.140	0.0851	0.0547
1node(25.4GB)	5	41	66.726	40.893, 12.606	0.0694	0.0578	0.0113

Table 7.3: Task 2's results (1node[:40] dataset: 10 reducers and 1node dataset: 50 reducers).

Data Set	Num of keys/mapper	Num of mappers	Total execution time(s)	Stage execution times(s)	Total cost($)	Lambda cost($)	S3 cost($)
1node[:40](5GB)	1	90	86.757	13.616, 24.017 12.861, 7.378, 0.317, 0.565	0.0660	0.0533	0.0125
1node[:40](5GB)	3	30	122.219	38.843, 23.299, 12.986, 7.321, 0.300, 0.629	0.0575	0.0513	0.00604
1node(25.4GB)	1	252	135.271	27.017, 21.511, 21.584, 9.154, 0.188, 1.517	0.496	0.302	0.192
1node(25.4GB)	3	84	174.706	59.566, 18.217, 21.587, 9.256, 0.185, 2.003	0.352	0.249	0.102

Table 7.4: Task 3's results (1node[:40] dataset: 20, 20 and 1 reducers and 1node dataset: 100, 100 and 1 reducers). Note that the dataset 1node/rankings is used in conjunction with the varying dataset uservisits.

Data Set	Num of keys/mapper	Num of mappers	Total execution time(s)	Stage execution times(s)	Total cost($)	Lambda cost($)	S3 cost($)
1node(1.28GB)	1	50	47.652	42.293	0.0532	0.0529	0
1node(1.28GB)	5	10	212.283	210.080	0.0524	0.0523	0
1node(1.28GB)	10	5	424.272	419.317	0.0524	0.0523	0
5nodes(6.38GB)	1	100	107.972	104.644	0.262	0.262	0
5nodes(6.38GB)	4	25	422.077	418.354	0.262	0.262	0

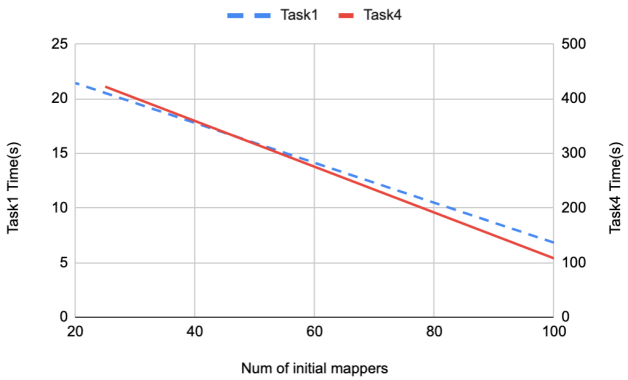
Table 7.5: Task 4's results.

Data Set	Num of keys/mapper	Num of mappers	Total execution time(s)	Stage execution times(s)	Total cost($)	Lambda cost($)	S3 cost($)
1node[:40](5GB)	1	40	341.105	95.495, 233.465	0.187	0.183	0.00327
1node[:40](5GB)	5	8	714.351	470.255, 232.758	0.182	0.181	0.000685
1node(25.4GB)	1	202	204.862	98.403, 89.553	0.665	0.609	0.0547
1node(25.4GB)	5	41	581.389	463.708 85.851	0.594	0.583	0.0113

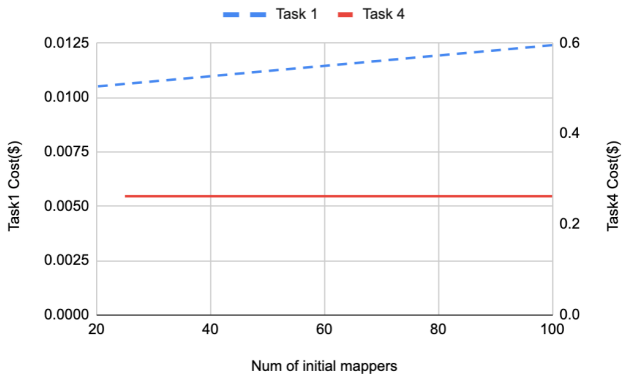
Table 7.6: Task 5's results (1node[:40] dataset: 15 reducers and 1node dataset: 50 reducers).

Data Set	Num of keys/mapper	Num of mappers	Total execution time(s)	Stage execution times(s)	Total cost($)	Lambda cost($)	S3 cost($)
1node[:40](5GB)	1	90	986.229	44.628, 373.842, 50.672, 437.585, 30.384, 1.192	0.791	0.770	0.0202
1node[:40](5GB)	3	30	1081.754	128.664, 372.983, 50.385, 441.560, 30.373, 0.765	0.779	0.769	0.0105
1node(25.4GB)	1	252	984.590	55.927, 127.855, 60.761, 648.187, 30.367, 1.821	2.716	2.522	0.192
1node(25.4GB)	3	84	1146.283	149.741, 127.027, 61.541, 654.338, 30.355, 2.979	2.602	2.500	0.102

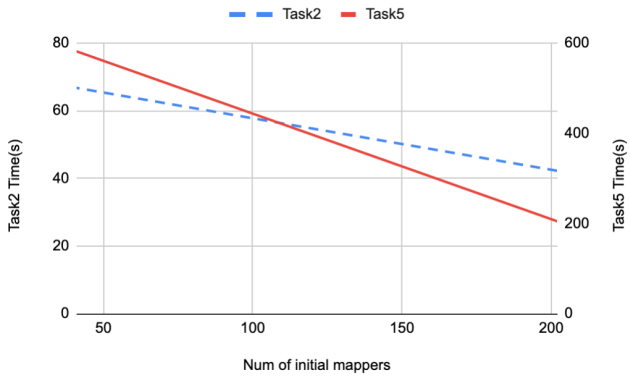
Table 7.7: Task 6's results (1node[:40] dataset: 30, 30 and 1 reducers and 1node dataset: 100, 100 and 1 reducers). Note that the dataset 1node/rankings is used in conjunction with the varying dataset uservisits.



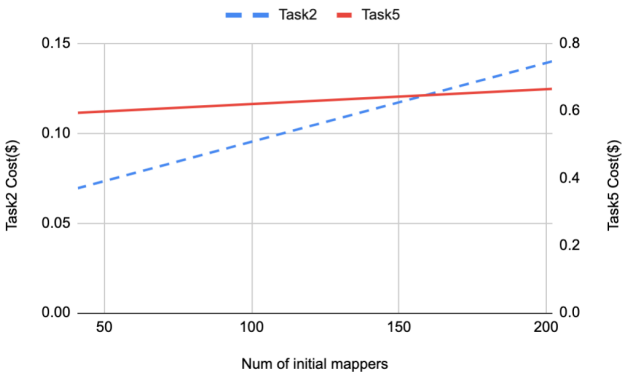
(a) Execution Time(s): Task 1 and Task 4



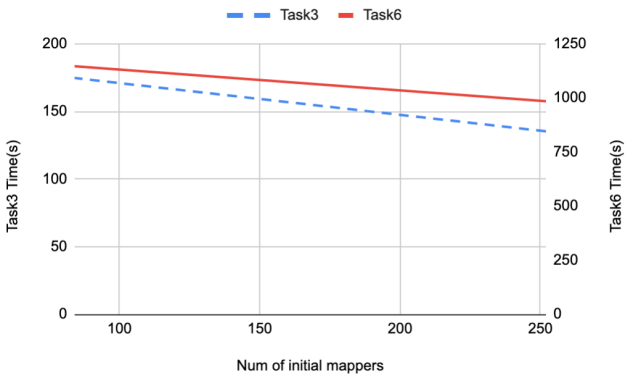
(b) Cost($): Task 1 and Task 4



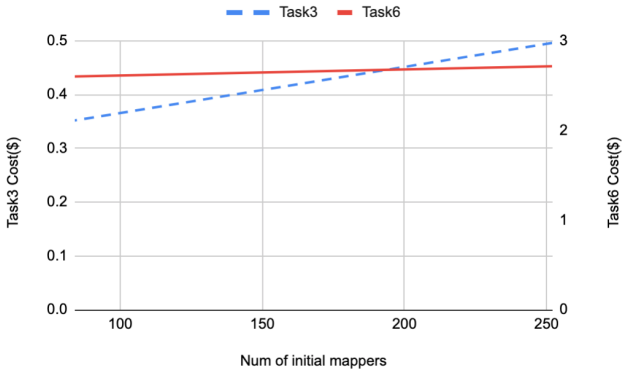
(c) Execution Time(s): Task 2 and Task 5



(d) Cost($): Task 2 and Task 5



(e) Execution Time(s): Task 3 and Task 6



(f) Cost($): Task 3 and Task 6

Figure 7.1: Left graphs: Num of initial mappers vs Execution times(s). Right graphs: Num of initial mappers vs Costs($). For each task, its metrics on the large dataset are used.

Varying number of reducers

Num of Reducers	Total execution time(s)	Stage execution times(s)	Total coordinator execution time(s)	Total cost($)	Lambda cost($)	S3 cost($)
45	41.619	12.198, 16.839	14.551	0.131	0.0809	0.0493
50	42.227	12.879, 15.739	15.452	0.140	0.0851	0.0547
75	48.750	14.823, 12.153	18.249	0.180	0.0975	0.0820
100	49.050	18.447, 10.257	21.580	0.229	0.119	0.109

Table 7.8: Task 2's results with the dataset 1node and 1 input key per initial mapper.

Num of Reducers	Total execution time(s)	Stage execution times(s)	Total coordinator execution time(s)	Total cost($)	Lambda cost($)	S3 cost($)
100	135.271	27.017, 21.511, 21.584, 9.154, 0.188, 1.517	60.521	0.496	0.302	0.192
125	154.145	29.600, 18.795, 21.611, 8.680, 0.174, 2.772	74.905	0.601	0.342	0.257
150	167.242	33.596, 18.073, 21.811, 8.290, 0.191, 3.713	87.760	0.735	0.404	0.329
200	187.344	38.468, 16.040, 22.406, 8.382, 0.211, 3.985	117.045	0.974	0.480	0.492

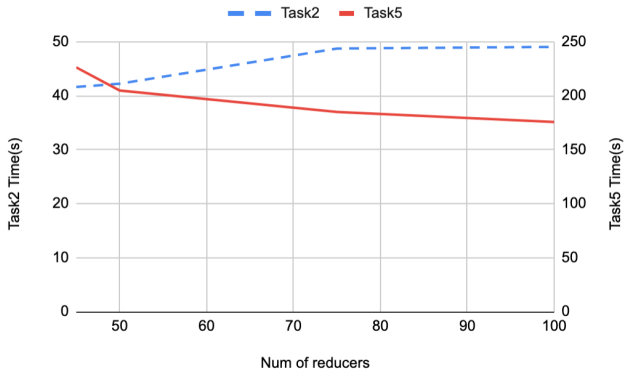
Table 7.9: Task 3's results with the dataset 1node and 1 input key per initial mapper.

Num of Reducers	Total execution time(s)	Stage execution times(s)	Total coordinator execution time(s)	Total cost($)	Lambda cost($)	S3 cost($)
45	226.219	98.290, 99.485	15.347	0.659	0.609	0.0493
50	204.862	98.403, 89.553	16.270	0.665	0.609	0.0547
75	184.980	101.762, 61.544	19.056	0.713	0.630	0.0820
100	175.735	103.756, 47.269	21.826	0.753	0.643	0.109

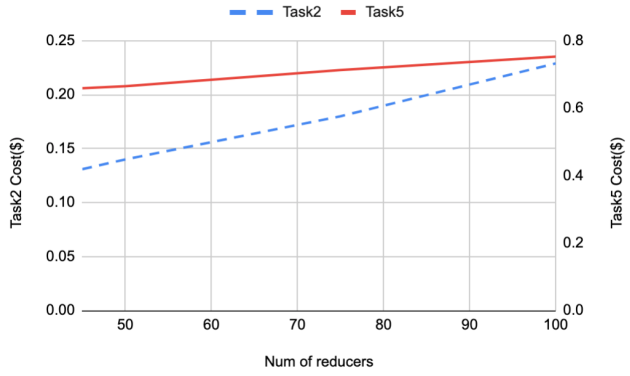
Table 7.10: Task 5's results with the dataset 1node and 1 input key per initial mapper.

Num of Reducers	Total execution time(s)	Stage execution times(s)	Total coordinator execution time(s)	Total cost($)	Lambda cost($)	S3 cost($)
100	984.590	55.927, 127.855, 60.761, 648.187, 30.367, 1.821	63.556	2.716	2.522	0.192
125	854.666	59.297, 104.196, 62.143, 520.979, 30.379, 2.753	80.169	2.878	2.619	0.257
150	783.895	63.101, 93.600, 63.893, 437.275, 30.357, 2.728	93.663	3.075	2.745	0.329
200	657.140	68.407, 69.424, 62.194, 328.071, 30.358, 4.231	124.028	3.379	2.885	0.492

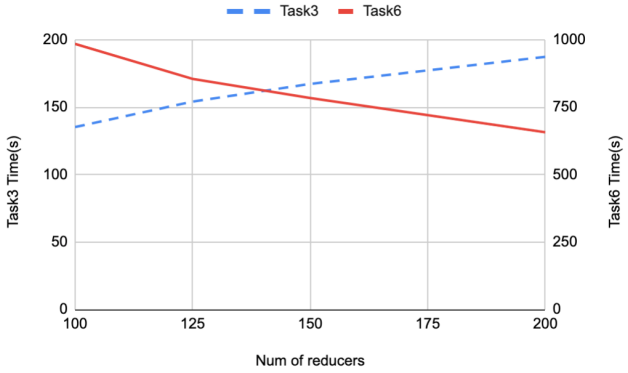
Table 7.11: Task 6's results with the dataset 1node and 1 input key per initial mapper.



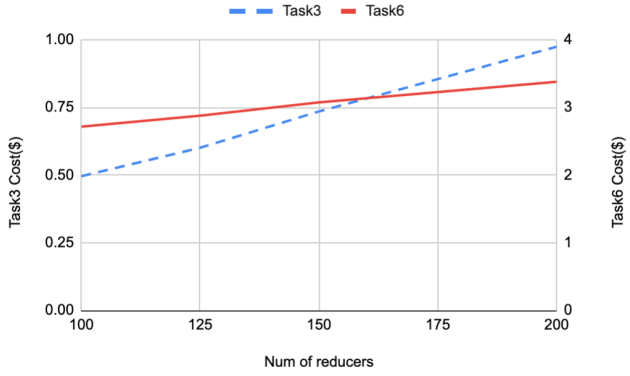
(a) Execution Time(s): Task 2 and Task 5



(b) Cost($): Task 2 and Task 5



(c) Execution Time(s): Task 3 and Task 6



(d) Cost($): Task 3 and Task 6

Figure 7.2: Left graphs: Num of reducers vs Execution times(s). Right graphs: Num of reducers vs Costs($).

7.1.3 Analysis 1: Number of initial mappers

Results of all tasks (shown in Tables 7.2, 7.3, 7.4, 7.5, 7.6 and 7.7) and Figure 7.1 suggest that by scheduling more mappers in the initial stage (allocating less input keys to each initial mapper), the execution time of a task decreases at the price of an increased cost. In a perfect world, one would expect that the cost of running a task with more initial mappers to be the same as the cost of running it with less. This is because for instance, if there are 10 input keys and each key takes 100s to be processed, having 10 mappers (each one processing one key) would yield a total Lambda execution time of 1000s which is the same as when only one mapper processes those 10 keys. Since the cost of a Lambda invocation is mainly based on its duration (as its request cost is negligible), same total Lambda execution times would yield the same cost. However, this is not the case due to the two following reasons: mapper and shuffling overheads.

A mapper has start-up overhead such as loading its map function from the pickle file, loading the configuration file, initialising different objects, etc. The existence of such an overhead can be verified by computing the average stage 1 execution time per input

key for tasks 1 and 4 since they are map-only tasks, hence are not subject to the shuffling overhead. Using Table 7.2, for task 1 on the dataset 1node with 1 key per mapper, the average execution time per key in stage 1 is $2.098/1 = 2.098$s. With 5 and 10 keys per mapper, this value drops to $8.662/5 = 1.732$s and $16.745/10 = 1.675$s respectively. Using Table 7.5, for task 4 on the dataset 1node with 1, 5 and 10 keys per mapper, the average execution times per key in stage 1 are 42.294s, 42.016s and 41.932s respectively. From these observations, we can deduce that this variable increases as more initial mappers are scheduled (allocating less keys to each initial mapper). This behaviour can also be observed on the results of the two tasks with their large datasets. It indicates the presence of mapper start-up overhead since the more initial mappers are scheduled, the more times this overhead will be experienced and hence, the larger the average execution time per key of a map stage will be.

For the short-running tasks, this mapper overhead is also the main reason why there is not linear scalability between number of initial mappers and total execution time. For instance, for task 1 on the dataset 1node, increasing the number of initial mappers from 10 to 50 should lead to a decrease of the total execution time by a factor of 5, however, it only drops by a factor of $10.907/3.666 = 3.0$. This start-up overhead is roughly constant per mapper regardless of the type of task. Hence, it does not affect the long-running tasks as much since the overhead of having more parallel mappers becomes insignificant when each input key takes longer to be processed. For instance, for task 4 on the dataset 1node, the total execution time drops by a factor of 4.5 when its number of initial mappers is increased from 10 to 50. Furthermore, for the long-running tasks, difference in the Lambda costs introduced by this overhead also becomes negligible compared to the overall Lambda cost. This is shown by results of tasks 4, 5 and 6 in their respective tables and Figure 7.1.

The second overhead is caused by the shuffling stage. In ServerlessMR, as intermediate data need to be persisted to a S3 shuffling bucket, apart from the Lambda cost, tasks that have more than one stage also have the additional request and storage costs of S3 shuffling bucket. Since the number of intermediate files produced by a shuffling stage is the number of mappers (m) multiplied by the number of subsequent reducers (r), a rise in m will lead to an increase in the number of intermediate files by a factor of r. The request cost of shuffling bucket will go up substantially if the number of intermediate files is increased. This is why for tasks 2, 3, 5 and 6, when the number of initial mappers is increased, its S3 cost goes up significantly. Currently, S3 charges \$0.005s per 1000 PUT requests and \$0.0004 per 1000 GET requests [40].

7.1.4 Analysis 2: Number of reducers

As with the initial mappers, in a perfect world, increasing the number of reducers of a task would reduce its execution time and would not increase its cost. However, in practice, the cost always goes up due to different overheads introduced by having more reducers as shown by Figure 7.2. What is more, for the short-running tasks, the performance overheads can be large enough such that adding more reducers to a task

increases its execution time.

Firstly, increasing the number of reducers of a task also increases the number of intermediate files generated in the shuffling stage and hence, leads to a rise in the cost of its S3 shuffling bucket (the aforementioned shuffling overhead). Apart from cost, the number of intermediate files also affects mapper execution times. For tasks 2, 3, 5 and 6 (results shown in Tables 7.8, 7.9,7.10 and 7.11 respectively), we can observe that if the number of reducers in stage S is increased, then the average execution time of the mappers in stage (S - 1) goes up. This is expected as each mapper would have to write more intermediate files to the shuffling bucket. Secondly, a reducer also has start-up overhead such as loading its reduce function from the pickle file.

For tasks that have short-running reduce stages (less than around 30 seconds, based on the results), the two performance overheads can make the tasks' execution times rise when their numbers of reducers are increased. For instance, in task 2, when the number of reducers is increased from 50 to 100, the task's total execution time increased from 42.227s to 49.050s. The execution time of its reduce stage (stage 2) decreased by 5.482s (from 15.739s to 10.257s) while its map stage (stage 1) execution time increased by 5.568s (from 12.879s to 18.447s). Similar behaviour can also be observed in task 3. For tasks with longer-running reduce stages, by increasing their numbers of reducers, their total execution times decrease, which is the expected behaviour. Unlike short-running reduce stages, the performance gain by having more reducers in long-running reduce stages outweighs the overheads introduced. In our case, since a long-running task only differs from its corresponding short-running task in how each input record is processed, the performance overheads should be alike between the two. For instance, the performance overheads on tasks 2 and 5 should be similar. In task 5, when the number of reducers is increased from 50 to 100, the task's execution time dropped from 204.862s to 175.735s. The execution time of its reduce stage (stage 2) decreased by 42.284s while the execution time of its map stage (stage 1) increased by 5.353s (similar amount as task 2). The same behaviour can be seen between tasks 3 and 6.

Lastly, for all the tasks, we can observe that the total coordinator execution time increases as the number of reducers is increased. The reason for this is having to invoke more reducers. For tasks where there are more stages after a reduce one such as tasks 3 and 6, it is also because there will be more coordinator invocations since each reducer that is not in the last stage calls the coordinator Lambda function when it terminates. To summarise, the increased execution cost of a task when its number of reducers is increased is caused by the rise in its total coordinator execution time, S3 request cost, mapper execution times and reducer start-up overheads.

7.1.5 Conclusion

In conclusion, having more parallelism does not always speed up a task's execution due to the different overheads that are introduced by it. In all the 6 tasks that were

analysed, by having more initial mappers, their execution times always drop due to the mapper overhead being insignificant. However, we do believe that in extreme cases when there is only one record in a file and the task is short, then having multiple mappers, each one processing one file might be slower than having only one mapper processing all the files. Increasing the number of reducers introduces a larger overhead. In tasks with long-running reduce stages (more than around 30 seconds, based on the results), having more reducers always lowers their execution times. However, if a task has shorter-running reduce stages, then increasing the number of reducers can backfire and increase its execution time. Lastly, having more parallelism always increases the cost of a task due to the different overheads. Users can either optimise for cost or performance of a task by adjusting the two parameters: number of initial mappers and number of reducers. However, it is important to understand that there is not a positive correlation between cost and performance in all cases.

7.2 ServerlessMR vs Hadoop

In order to thoroughly compare ServerlessMR's performance against a cluster-based Big Data framework such as Hadoop, tasks 1, 2 and 3 from the previous section are used along with five additional tasks (all composed of the stages map→reduce) listed below[2]:

- **TASK 7 - PageRank component**: computes the inlink count for each URL in the dataset Documents. Inlinks of a site are links on other websites that send traffic to this site. It forms part of the PageRank algorithm as Google uses inlinks as 'votes' for a site. This job allows us to model ServerlessMR's performance on a real-world use case and is the kind of task that MapReduce framework is commonly used for.

- **TASK 8 - Sorting**: sorts the records by a column. It is one of the shuffle-intensive tasks from Intel Hi-Bench suite [18]. In our case, this task sorts sourceIPs in the dataset Uservisits using the adRevenue column.

- **TASK 9 - Long Fibonacci**: uses the dataset Uservisits and computes the following expression for each source IP X:

  $$f(X) = fib((\sum_{duration_i \in durations(X)} fib(duration_i + 6))\%10 + 5)$$

  where fib is the Fibonacci function.

- **TASK 10 - Long PageRank component**: Same as task 8, but both in the map and reduce functions, a loop from 0 to 10000 has been added. This extends the task execution time and would help us compare the performance and cost of ServerlessMR with Hadoop on long-running computationally expensive tasks.

[2]Source code of all Hadoop/Hive tasks used in this evaluation are on: `https://github.com/hanglili/Hadoop-performance-benchmark`

- **TASK 11 - Long Aggregation task**: Same as task 2, but this time, the query has been rewritten using map and reduce functions and is run on Hadoop rather than Hive. To increase the task execution time, a loop from 0 to 2500 is added to the map function and a loop from 0 to 100000 is added to the reduce function.

7.2.1 Setup

To obtain the performance and cost metrics of these tasks, ServerlessMR was set up using the method mentioned in the previous section. For Hadoop, the first 3 tasks were run on Hive, which is the data tool that runs on top of Hadoop for providing data query and analysis. For Hadoop/Hive, AWS EMR was used since it allows one to easily set up different numbers and types of instances in a cluster. In order to have a fair comparison between the two frameworks, same amount of computing resources should be allocated to both. According to [17], a Lambda function with a memory limit of 1536MB has approximately one vcore (virtual core - AWS computation unit) and a Lambda function with 3008MB has 2 vcores. For each of the tasks, the same number of vcores is roughly allocated to Hadoop and ServerlessMR. However, AWS does not disclose the type of underlying instances that power Lambda, thus this is the best that we can do to ensure that both frameworks are provided with the same hardware resources. The configuration parameters used by ServerlessMR and Hadoop for each task are shown in Tables 7.12 and 7.13 respectively. The type of instance used by Hadoop is EC2 m4.4xlarge, that has 16vcores.

Task	Num of keys/mapper	Num of initial mappers	Num of reducers	Lambda Memory Limit (MB)
1	1	100	N/A	1536
2	1	202	50	1536
3	1	252	stage 2: 100, stage 4: 100, stage 6: 1	1536
7	1	109	20	3008
8	1	202	50	1536
9	1	202	100	1536
10	1	109	100	2048
11	1	202	100	1536

Table 7.12: Configuration parameters of each task in ServerlessMR.

Task	Num of instances	Cluster setup time(s)	Cluster teardown time(s)
1	7	456	94
2	14	535	91
3	16	514	123
7	15	236	108
8	14	223	71
9	14	398	77
10	15	302	127
11	14	245	93

Table 7.13: Configuration parameters and cluster setup and teardown times of each task in Hadoop/Hive.

7.2.2 Performance Analysis

		ServerlessMR				Hadoop/Hive			
Task	**Data Set**	**Execution time(s)**	**Setup time(s)**	**Teardown time(s)**	**Total Cost($)**	**Execution time(s)**	**Setup + Teardown time(s)**	**Total Cost($)**	**Total Cost with EMR ($)**
1	5nodes/rankings (6.38GB)	6.832	18.254	2.370	0.0124	74.473	21.161	0.143	0.186
2	1node/uservisits (25.4GB)	42.227	9.860	3.058	0.140	96.067	21.646	0.355	0.461
3	1node/rankings (1.28GB) + 1node/uservisits (25.4GB)	135.271	29.457	3.256	0.496	88.464	17.541	0.377	0.490
7	1node/crawl (29.0GB)	34.656	20.550	2.955	0.0911	100.421	11.579	0.373	0.485
8	1node/uservisits (25.4GB)	55.481	23.909	2.418	0.148	83.632	14.368	0.305	0.396
9	1node/uservisits (25.4GB)	238.327	20.344	1.955	0.948	101.324	8.676	0.342	0.444
10	1node/crawl (29.0GB)	477.422	25.871	2.273	1.048	107.496	8.504	0.387	0.503
11	1node/uservisits (25.4GB)	214.017	25.252	3.033	0.913	95.866	6.134	0.317	0.412

Table 7.14: ServerlessMR and Hadoop performance and cost metrics.

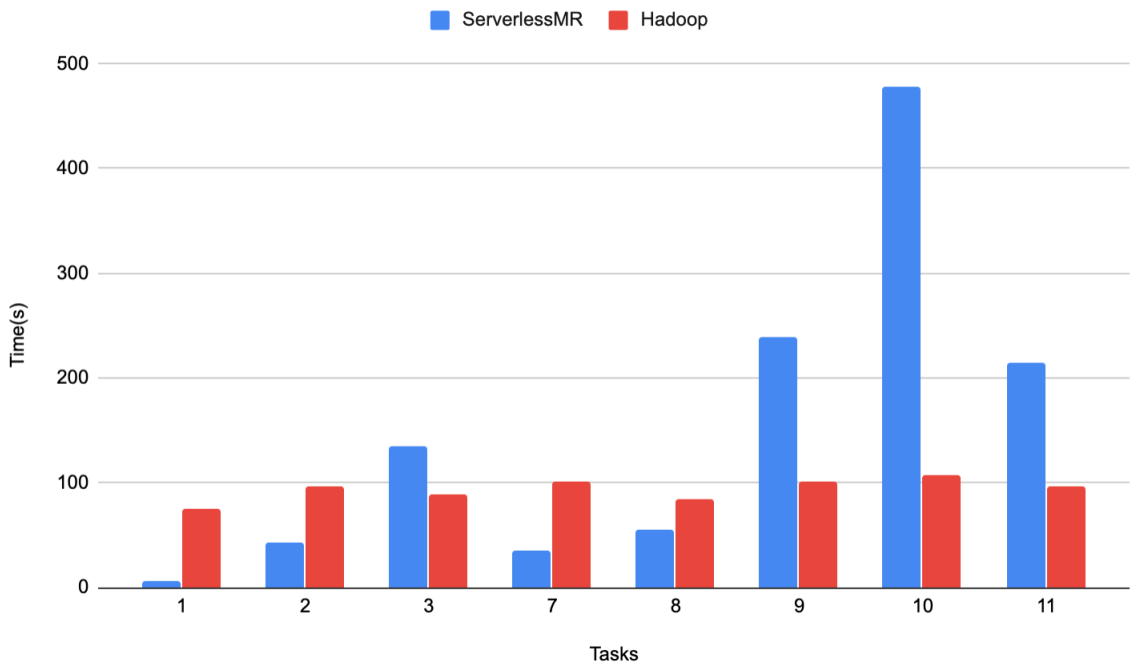


Figure 7.3: Execution Times(s) of ServerlessMR and Hadoop.

The execution time and cost of each task on both ServerlessMR and Hadoop/Hive are shown in Table 7.14. The performance comparison between both frameworks is il-

lustrated in Figure 7.3. This comparison puts Hadoop/Hive in the best possible light since the cluster start-up (between 5 and 10 minutes) and tear-down (between 2 and 3 minutes) times are not included in its setup + teardown time column in the table. For a task in Hadoop/Hive, this column only includes the time taken for the task to be set up and tear down. For Hive, this also covers the time spent (about 2 or 3 seconds) on creating an external table that maps to the input data in S3. Note that all tasks have been executed on large data sets, making them data-intensive jobs.

Considering the task execution, setup and teardown times altogether, we can see that ServerlessMR is both cheaper and faster than Hadoop in 4 out of the 8 tasks. These 4 tasks have the properties of being short and non-computationally expensive. We suspect that a reason for the faster execution is that Hadoop has some additional start-up overhead when a task is first started. For Hive queries, this overhead might be even larger, since queries have to be transformed into map and reduce functions. This would justify why task 11 is more performant than task 2, even though task 11 is a more computationally-expensive version of task 2. However, for the longer-running tasks such as 9, 10 and 11, Hadoop is superior both in terms of speed and cost. Upon investigation, this is due to two main reasons:

- Hadoop can start running the reduce phase before all mappers finish. The reduce phase in Hadoop has 3 steps: shuffle, sort and reduce [24]. The shuffle step which consists of the reducers collecting data from the mappers, is the step that can be executed before the full termination of mappers. The other two steps have to wait. This allows Hadoop to speed up the overall computation time, but also spreads out data transfer from mappers to reducers over time, which avoids creating network bottlenecks. For example, in task 10, Hadoop already achieved a progress of 24% in the reduce phase by the time all its mappers finished. This optimisation of starting reduce phase earlier is not present in ServerlessMR and cannot be easily implemented because of two limitations of serverless functions. Firstly, a reducer Lambda cannot directly collect data from the mappers since two Lambda invocations cannot communicate with each other without any intermediate systems or storage. Secondly, serverless functions have execution time limits. If a reducer Lambda starts its execution earlier, then later it would have less time to apply the reduce function.

- In each of the long-running tasks, most of its stages take more time in ServerlessMR. This is due to two reasons. Firstly, the duration of a stage in ServerlessMR is determined by the longest executor in that stage and thus, the execution time of a task is determined by the longest executors in all its stages. For instance, in extreme cases such as task 10, there is a time difference of 295s between first mapper that finished and the last. This makes ServerlessMR less performant for tasks with datasets that lead to an uneven workload among executors. Currently, for an S3 input storage, each input key (S3 object key) is assigned to a mapper. If one object requires more processing than others, then its assigned mapper is going to take longer than others. On the other hand, Hadoop tries to balance the workload as much as possible by splitting even a single S3 object into smaller

chunks that can be processed by different mappers [52].

Secondly, in most cases of the long-running tasks, the average run-time of an executor in a stage is larger in ServerlessMR. For example, for task 10, the average execution time of a mapper in ServerlessMR is around 221s whereas in Hadoop, each mapper has an average execution time of less than 60s. The reason behind this could be that the type of machines (m4.4xlarge) used for the Hadoop cluster is more powerful than the underlying instances running AWS Lambda. This cannot be verified since AWS provides very little documentation about the exact instance type that is running the Lambdas. Apart from that, data locality could also play a role here. Since each worker node in a Hadoop cluster is also a data node and stores the intermediate data, Hadoop assigns a task to a node that stores or is close to the data. This avoids network bottlenecks and having to read from a remote storage, hence decreases the average run-time of each stage.

In ServerlessMR, there are several ways to improve the run-time of a stage. One is adding more parallelism to each stage by breaking down an S3 object (input key) into smaller chunks and scheduling more mappers since the mapper overhead is insignificant. Each mapper would then process a small chunk of an object. Another solution is to keep a dynamic timeout threshold for every stage. Once the threshold of a stage is reached, every executor in that stage invokes another executor to share its workload. These suggestions could be future extensions to improve ServerlessMR performance.

Task 3 is special since it requires a join operation on two different data sets. On this task, Hive performs better than ServerlessMR. We suspect the reason to be the optimisation that Hive performs on the query execution plan. In ServerlessMR, task 3 is implemented as a multi-stage MapReduce task that comprises of map, reduce, map, reduce, map and reduce stages. However, on Hive, this query is transformed into a Hadoop task, consisted of only 5 stages.

In most of the tasks, Hadoop's task setup and teardown times are smaller. However, this could be due to the fact that on ServerlessMR, the tasks were set up using a laptop (that has two cores and a RAM of 8GB), which is far less powerful than a m4.4xlarge machine. In addition to that, the network connection could also play a role here.

7.2.3 Cost Analysis

The cost comparison is illustrated in Figure 7.4. To put Hadoop/Hive in the best possible light, its cost used for comparison is the one that excludes the EMR cost. For short-running tasks such as 1, 2, 7 and 8, ServerlessMR can cost up to 11.5 times less than Hadoop. The reason for this cost difference might be related to ServerlessMR's shorter execution times on these tasks as they happen to be the only ones in which it performs better. However, for longer-running tasks, Hadoop can be up to 2.8 times cheaper, as shown by tasks 9, 10 and 11. This seems to be a reasonable factor. Lambda

costs around 0.000025 per second per vcore whereas a m4.4xlarge EC2 instance costs 0.0000139 per second per vcore. This means that for an equal amount of computing resources, Lambda is approximately twice more expensive. It makes sense that Lambda has a larger per-unit resource cost, which corresponds to the price for characteristics such as on-demand invocation, elasticity, etc. Therefore for a task, even if Hadoop and ServerlessMR spend an equal amount of computing resources, ServerlessMR would still be at least twice more expensive than Hadoop. The remaining difference in the cost is most likely due to the S3 cost for requests and storage of intermediate data.

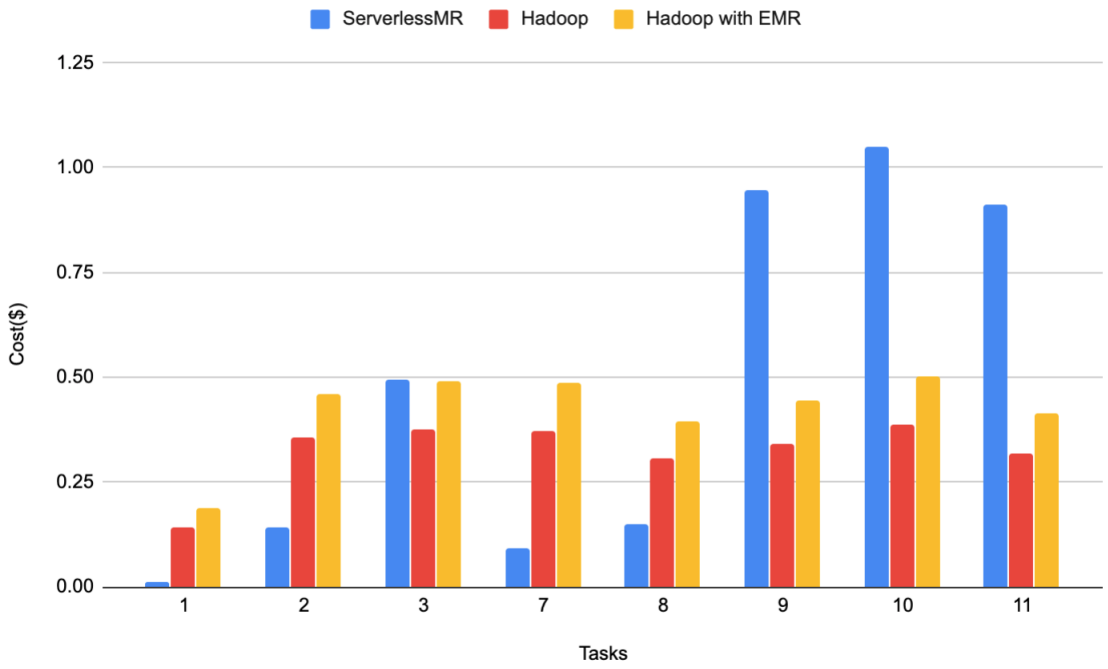


Figure 7.4: Costs($) of ServerlessMR and Hadoop (with and without EMR).

An interesting question to answer is what is the minimum number of times a task has to be executed in a period of time such that having a Hadoop cluster for that time period would cost less than running it on ServerlessMR. To answer this question, we can first work out the minimum utilisation of the cluster in order for the cluster's existence to be justified. Assuming that only 1 task (T) is run in both frameworks, then this minimum utilisation threshold is calculated as execution cost of T in Hadoop divided by the execution cost of T in ServerlessMR. The derivation for the aforementioned equation is in Figure 7.5.

$$\text{Min num of executions required in P} = \frac{\text{Cluster cost in P}}{\text{T's execution cost in ServerlessMR}} \tag{7.1}$$

$$\text{Min Utilisation} = \frac{\text{Min total busy time of the cluster}}{\text{Total time of the cluster}} \tag{7.2}$$

$$= \frac{(\text{Min num of executions in P}) * (\text{T's execution time in Hadoop})}{\text{Number of seconds in P}} \tag{7.3}$$

$$= \frac{(\text{Cluster cost in P}) * (\text{T's execution time in Hadoop})}{(\text{T's execution cost in ServerlessMR}) * (\text{Number of seconds in P})} \tag{7.4}$$

$$= \frac{(\text{Cluster cost per second}) * (\text{T's execution time in Hadoop})}{(\text{T's execution cost in ServerlessMR})} \tag{7.5}$$

$$= \frac{\text{T's execution cost in Hadoop}}{\text{T's execution cost in ServerlessMR}} \tag{7.6}$$

where

$$\begin{aligned} T &= \text{the task to be executed} \\ P &= \text{time period} \end{aligned}$$

Figure 7.5: Minimum Utilisation Derivation.

The minimum number of executions required in time period P can also be expressed in terms of the minimum utilisation:

$$\text{Min num of executions in P} = \frac{(\text{Min utilisation}) * (\text{Number of seconds in P})}{(\text{T's execution time in Hadoop})} \tag{7.7}$$

The above equations can also be extended to cases when there are multiple tasks $(T_1...T_n)$ to be executed, by substituting T's execution time and cost with the weighted averages of $(T_1...T_n)$ execution times and costs. The weights are based on the execution frequency ratios of tasks. For instance, if there are two tasks T_1 and T_2 to be executed with ratios of 1/4 and 3/4 respectively[3], then T_1's weight will be 0.25 and T_2's weight, 0.75. The proof for this extended equation is shown in Figure A.1 in Appendix.

For the following experiments, we assume that users only run longer-running tasks, since for short-running tasks (less than around 2 minutes, based on the results), ServerlessMR is cheaper. For this reason, we only answer the aforementioned question for tasks 9, 10 and 11. Using the equations in (7.6) and (7.7), the following results are produced:

- **Task 9**: min utilisation is 0.342 / 0.948 = **36.1%** and min num of executions required (in a day) = (0.361 * 3600 * 24) / (101.324 + 8.676) = **284**.

[3]This means that on average out of 4 executions, T_1 is executed once and T_2, three times.

- **Task 10**: min utilisation is 0.387 / 1.048 = **36.9%** and min num of executions required (in a day) = (0.369 * 3600 * 24) / (107.496 + 8.504) = **275**.
- **Task 11**: min utilisation is 0.317 / 0.913 = **34.7%** and min num of executions required (in a day) = (0.347 * 3600 * 24) / (95.866 + 6.134) = **294**.

From a user's perspective, it would be more interesting to answer the aforementioned question when both frameworks have similar response times for a task. For this reason, tasks 9, 10 and 11 were executed again using a less powerful Hadoop cluster (using less m4.4xlarge machines). The results are shown in Table 7.15. Performing the same experiment as before with these new figures produce the following results:

- **Task 9**: min utilisation is 0.249 / 0.948 = **26.3%** and min num of executions required (in a day) = (0.263 * 3600 * 24) / (215.452 + 8.548) = **102**.
- **Task 10**: min utilisation is 0.312 / 1.048 = **29.8**% and min num of executions required (in a day) = (0.298 * 3600 * 24) / (458.594 + 9.406) = **56**.
- **Task 11**: min utilisation is 0.242 / 0.913 = **26.5%** and min num of executions required (in a day) = (0.265 * 3600 * 24) / (210.129 + 7.871) = **106**.

Task	Num of instances	Total Execution time(s)	Set up + Tear down time(s)	Total Cost($)	Total cost with EMR($)
9	5	215.452	8.548	0.249	0.324
10	3	458.594	9.406	0.312	0.406
11	5	210.129	7.871	0.242	0.315

Table 7.15: Results and cluster parameters of Hadoop tasks that yield similar response times as their ServerlessMR's executions.

In each of the experiments, the minimum utilisation thresholds of all the analysed tasks are close to each other. For the first experiment, the average minimum utilisation of the three tasks is 35.9% whereas for the second second experiment, this value dropped to 27.5%. This reduction is due to the decrease in the Hadoop execution cost of each task. This suggests that a less powerful cluster will have a smaller minimum utilisation. Apart from that, in both experiments, the minimum number of executions required are similar for tasks 9 and 11. This is expected because they have the same cluster cost, similar Hadoop execution times, similar Hadoop and ServerlessMR execution costs. Unlike in the first experiment, task 10's minimum number of executions is quite different to the other two in the second experiment. This is because task 10's ServerlessMR execution time is greatly affected by the issue of unbalanced workload, making it inappropriate for the second experiment.

In 2017, AWS has changed the pricing model of EC2 instance from charging per hour to charging per second [48]. This might make you wonder why is ServerlessMR needed then since AWS also offers per-second billing of EC2 instances. However, a lot of time is still wasted on cluster initialisation every time a task/query needs to be run. If users want to avoid this start-up overhead, then they will need to have the cluster running all the time. This is one of the reasons why organisations still choose to have cluster instances running the entire time even when the cluster is idle.

7.2.4 Conclusion

Performance and cost analyses were carried out on the results obtained from 8 data-intensive tasks. From these analyses, given roughly the same amount of resources to both frameworks, per execution, it seems that ServerlessMR is better in short-running tasks (less than around 2 minutes, based on the results) both in terms of performance and cost. However, for longer-running tasks where more computationally-expensive operations are required, Hadoop is cheaper per execution and more performant. Even though it is cheaper, the cluster initialisation can take a long time (could be up to 10 minutes). To avoid this start-up overhead every time a task is run, users might choose to have a Hadoop cluster running all the time. Whether the cost of this cluster can be justified is dependent on the utilisation of the cluster. If a cluster utilisation is larger than a minimum threshold, then it is worthwhile having the cluster. This minimum threshold for a task T can be calculated as the execution cost of T in Hadoop divided by the execution cost of T in ServerlessMR. For instance, using the settings where both frameworks yield similar execution times, for tasks 9, 10 and 11, having a Hadoop cluster would be cheaper than ServerlessMR if its utilisation exceeds around 28%.

To conclude, users should use ServerlessMR for short-running tasks (less than around 2 minutes). For longer-running tasks, Hadoop should be employed if job completion times are critical and the cluster has a higher utilisation than its minimum threshold. This minimum threshold is most accurate if it is calculated using the aforementioned equation on the tasks to be executed, however, if that is not possible, it can be estimated to be around 30%. As Lambda is about twice as expensive as on-demand instances per GB-hour of execution, even if a serverless data framework spends an equal amount of computing resources as Hadoop on a task, it would still cost twice as much as Hadoop, yielding a minimum utilisation threshold of 50%. Then adding the cost of remote storage which is required by any serverless data frameworks, this threshold will decrease. Hence, we believe that this estimate of the minimum utilisation threshold (30%) is broadly applicable to other serverless Big Data frameworks too. Apart from that, note that if a more powerful Hadoop cluster is used, then this minimum utilisation threshold will increase as the task execution cost rises.

Other serverless MapReduce frameworks such as Corral and AWS MapReduce have also been tested using some of the tasks from the Berkeley Big Data Benchmark suite. In tasks that have been both evaluated in ServerlessMR and these two frameworks, ServerlessMR has shown to be the most performant out of the three[4].

7.3 Local Execution

To evaluate the local execution feature, the behaviours of the local and cloud executions of 3 different jobs are compared, with the aim for them to be the same. Apart

[4]For Corral and AWS MapReduce performance results, check these two URLs respectively: `https://github.com/bcongdon/corral/tree/master/examples` and `https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/`

from that, the execution times will also be compared and it is expected that jobs execute faster on the cloud since local Lambda functions run as containers on a local machine.

ServerlessMR's local and cloud executions are tested for identical outputs using a wide variety of tasks: map-only, single map-reduce and multi-stage map-reduce tasks. The three tasks used are the aforementioned tasks 1, 2 and 3. Each task's configuration is shown in Table 7.16 along with its execution, setup and teardown times in both local and cloud environments. Note that a tiny version of each dataset is used because otherwise locally-deployed tasks can take long to execute. Once these tasks are executed locally and on the cloud, their output files are compared using their MD5 hash checksums.

			LOCAL			CLOUD		
Task	**Data Set**	**Num of reducers**	**Execution time(s)**	**Setup time(s)**	**Teardown time(s)**	**Execution time(s)**	**Setup time(s)**	**Teardown time(s)**
1	tiny/rankings (77.6KB)	N/A	15.610	3.917	0.142	1.386	10.437	2.891
2	tiny/uservisits (1.7MB)	5	42.597	5.234	0.214	2.840	12.168	3.738
3	tiny/rankings (77.6KB) + tiny/uservisits (1.7MB)	2: 5, 4: 5, 6: 1	123.678	9.596	0.234	8.631	23.659	3.833

Table 7.16: Results of local and cloud executions of tasks 1, 2 and 3 (For all tasks, number of keys per initial mapper = 1).

7.3.1 Comparison on behaviour

For all three tasks, the outputs produced by their local and cloud executions have the same hash values, with a few exceptions which are caused by the inconsistency of floating point results. For instance, one output is 0.9012531900000001 in one environment while in another environment, it is 0.90125319. Taking this inconsistency into consideration, for all three tasks, output files produced by their local and cloud executions are equivalent. Thus, we conclude that the local execution of a task behaves in the same way as its cloud execution.

7.3.2 Comparison on setup, execution and teardown times

As expected, a task that is executed in the cloud has a smaller execution time. For all the tasks analysed, their cloud executions can be up to 14 times faster. We believe that for longer running jobs, the difference will be even greater. However, the setup and teardown times of a local execution is about 2-3 times faster. This difference is probably due to the latency introduced by the network communication required for sending requests to AWS. For each of the tasks, its overall time (sum of the setup, execution and teardown times) is shorter when it is executed on the cloud environment.

7.3.3 Conclusion

The local execution of a job successfully simulates the behaviour when it is run on the cloud, in all three types of tasks. However, tasks executed locally can take longer due to having only one machine (the local machine) performing the computation. This does not take away the fact that local execution allows users to easily debug and test a task before deploying it to the cloud. When a task is executed locally, logs from different Lambdas can be streamed at the same time to one centralised location, which helps one to track down an issue very quickly. On the cloud, debugging is harder because one often has to go through different log groups and search for errors, since each stage of a task has its own Lambda function and hence its own log group. Apart from that, testing a task on the cloud costs money. Hence even though, the overall time taken by a task is shorter when it is executed on the cloud, the local execution feature still serves its purpose. Besides, local execution times are expected to be short since locally, tasks are intended to be tested against a small dataset (which is representative of the real data). As a future extension, local testing can be added as an extra step in the CD process of a task's deployment where the task is automatically tested in a local environment before deploying it to the production.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

We have presented ServerlessMR, a Big Data MapReduce framework that allows for the distributed processing of large data sets on a fully serverless architecture. ServerlessMR removes the high barrier to the usage of Big Data frameworks, created by the complex cluster setup, management and configuration in server-oriented frameworks such as Hadoop and Spark. It inherits the simplicity of serverless model, thus resource provisioning is completely abstracted away from users. The high elasticity of serverless computing allows ServerlessMR to efficiently support jobs whose resource requirements vary significantly during their execution, achieving auto-scaling naturally. Apart from that, it offers a truly pay-as-you-go model where users only get charged for the duration of a job execution.

Compared to other existing serverless Big Data frameworks, ServerlessMR offers the following innovative features:

- **Support for multi-stage and multi-pipeline tasks:** even though ServerlessMR operates on a restrictive execution environment (AWS Lambda), it supports multi-stage (section 5.6) and multi-pipeline (section 5.7) tasks such as map-reduce-map-reduce-map.
- **Completely serverless execution:** unlike other serverless data frameworks which still require at least one long-running server in a job execution, ServerlessMR jobs can be executed in a completely serverless fashion (section 5.2). Furthermore, through its web application, jobs can even be created, set up and modified within a fully serverless environment (sections 5.8.7 and 5.8.8).
- **Easier testing of data processing jobs:** thanks to its local execution feature (section 5.3), testing of data processing jobs can be performed easily on a local machine.

Through this project, we have demonstrated that a completely serverless Big Data framework is possible. In short tasks (less than around 2 minutes), such a framework is both cheaper per execution and faster than Hadoop, when roughly the same

amount of resources is allocated to both. However, for longer-running tasks where more computationally-expensive operations are required, Hadoop is cheaper per execution and more performant. A serverless data framework's cost can still be lower than a long-running Hadoop cluster if the cluster's utilisation is smaller than its minimum threshold. For tasks $T_1...T_n$ that are to be executed in a cluster, this threshold can be calculated using both the weighted averages of execution costs of $T_1...T_n$ in the serverless framework and Hadoop (formula shown in Figure A.1). For ServerlessMR and Hadoop, from our experimental results, this threshold is estimated to be around 30% for longer-running tasks. Due to the cost introduced by remote storage (required by any serverless data framework) and the fact that Lambda is about twice as expensive as on-demand instances per GB-hour of execution, we believe that this estimate is broadly applicable to other serverless data frameworks too. To conclude, a serverless Big Data framework is a great fit for:

- Short-running tasks.
- Ad-hoc analytics.
- Longer-running tasks if the cluster utilisation of a server-oriented framework is lower than its minimum threshold and task completion times are not critical.

We plan to submit these findings for publication to a conference later in the year.

8.2 Future Work

Limitations of ServerlessMR are discussed below along with some prominent future extensions. These limitations would have been addressed if we had more time.

- **Performance optimisation:** There are several ways to further optimise ServerlessMR's performance. Firstly, more parallelism could be added to each stage by breaking down each S3 object into segments and scheduling more executors, each processing a segment. Secondly, techniques such as speculative execution on a dataset sample could be used to balance the workload among executors since one of the current limitations is that the duration of a stage is determined by the longest executor in that stage.
- **Non-standard libraries and dependent source files:** Currently, ServerlessMR only uploads the source file where a provided function is defined, to an executor Lambda function. Hence all the code for each map/reduce function has to be in one file. An extension to remove this limitation is to collect all the required non-default Python libraries and user-defined modules for each of the user-provided functions. Then, source files of these modules are also uploaded to an executor Lambda function.
- **Lambda execution time limit:** Any Lambda function has an execution time limit of 15 minutes. For this reason, in ServerlessMR, a stage cannot be longer than 15 minutes. This AWS limitation can be overcome in the following way. Each

executor keeps its execution time and input pairs processed so far. When an executor's time is reaching 15 minutes, it will invoke a new executor passing the remaining input pairs to be processed as parameter, before terminating itself.

- **Scheduling heuristics:** In the current implementation, maximum parallelism is employed for any task, i.e., one executor is assigned to process one input file. However, more sophisticated heuristics that are custom to a job could be obtained using simulation and modelling techniques/tools such as JMT[1].
- **Shuffling technique:** Although the current shuffling technique works well, more advanced techniques can be explored. An idea is to combine S3 with a distributed caching system such as Redis [68] to speed up the shuffling phase.

[1] `http://jmt.sourceforge.net/`

Appendix A

Appendix

A.1 driver.json

The driver.json configuration file has the following json fields:

- `"region"(string)`: the region where the AWS services will defined. If not provided, the default value is ”us-east-1”.
- `"lambdaMemoryProvisioned"(integer)`: the memory size of the AWS Lambda to be provisioned. If not provided, the default value is 1536 MB.
- `"concurrentLambdas"(integer)`: the Lambda concurrency limit of an AWS account in each region. This concurrency limit determines how many function invocations can run simultaneously in one region . If not provided, the default value is set to 1000.
- `"lambdaReadTimeout"(integer)`: the time in seconds till a timeout exception is thrown when attempting to read from a connection. If not provided, the default is 300 seconds.
- `"botoMaxConnections"(integer)`: the maximum number of boto [1] connections to keep in a connection pool. If not provided, the default value of 1000 is used.

Note that all the fields in the driver configuration file are optional, then the file can be left empty as `{}` or omitted if the default configuration is suitable for your job.

A.2 static-job-info.json

The static-job-info.json configuration file has the following **compulsory** json fields:

- `"jobName"(string)`: the name of the job.
- `"lambdaNamePrefix"(string)`: the prefix that is added to the names of different Lambdas.

[1] **Boto**: AWS SDK for Python. For more information: `https://github.com/boto/boto3`.

- `"shufflingBucket"(string)`: the name of the S3 shuffling bucket used for this job.
- `"inputSourceType"(string)`: the input data storage type. Currently, the types supported are ”s3” and ”dynamodb”.
- `"inputSource"(string)`: the name of the data storage where the input data are stored.
- `"outputSourceType"(string)`: the output data storage type. Currently, the types supported are ”s3” and ”dynamodb”.
- `"outputSource"(string)`: the name of the data storage where the output data will be stored.
- `"useCombine"(boolean)`: a flag to denote whether a Combiner function is to be used.
- `"localTesting"(boolean)`: a flag to denote whether the job is to be deployed locally.
- `"serverlessDriver"(boolean)`: a flag to denote whether the driver is to be run on a serverless function (AWS Lambda).

The json fields listed below are **compulsory** depending on the other attributes:

- `"inputPrefix"(string)`: compulsory if `"inputSourceType"` is set to ”s3”. It specifies the prefix of the object keys in the S3 input bucket that corresponds to the input data. If all the objects in the provided S3 input bucket are inputs of the job, then this field can be set to `""`.
- `"inputPartitionKeyDynamoDB"(list)`: compulsory if `"inputSourceType"` is set to ”dynamodb”. It specifies the partition key name and type of the items in the input DynamoDB table. An example of this field is: `["recordId", "N"]`.
- `"inputSortKeyDynamoDB"(list)`: compulsory if `"inputSourceType"` is set to ”dynamodb” and the specified input DynamoDB table has a composite primary key (uses both partition and sort keys). It specifies the sort key name and type of the items in the input DynamoDB table. An example of this field is: `["timeProcessed", "N"]`.
- `"inputProcessingColumnsDynamoDB"(list)`: compulsory if `"inputSourceType"` is set to ”dynamodb”. It contains a list of attributes (name and type) that should be retrieved from the input table. These retrieved attributes will be present in the inputs to the user-defined functions. An example of this field is: `[["sourceIP", "S"], ["adRevenue", "N"]]`.
- `"outputPrefix"(string)`: compulsory if `"outputSourceType"` is set to ”s3”. It specifies the prefix to be added to the output object keys in the S3 output bucket. This field can be `""` if no prefix should be prepended to the output keys.

- `"outputPartitionKeyDynamoDB"(list)`: compulsory if `"outputSourceType"` is set to ”dynamodb”. This field specifies the partition key name and type of the DynamoDB output table. The output keys will be stored under this partition key attribute.
- `"outputColumnDynamoDB"(list)`: compulsory if `"outputSourceType"` is set to ”dynamodb”. It defines the attribute name and type in the DynamoDB output table. The output values will be stored under this attribute.
- `"inputColumnsDynamoDB"(list)`: compulsory if `"inputSourceType"` is set to ”dynamodb” and `"localTesting"` is set to True. It contains a list of attribute names and types that match to the schema of the local input csv files.
- `"localTestingInputPath"(string)`: compulsory if `"localTesting"` is set to True. It specifies the local directory path where the input data is. This data will be uploaded to the local S3 input bucket and serves as the input of the job. The path is relative to the working directory of the project where the job is defined.

Optional fields are listed below:

- `"optimisation"(boolean)`: a flag to denote whether the job is to be run in optimisation mode, i.e., without collecting metrics for Web UI.

A.3 Locally executed end-to-end tests

Listing A.1: An end-to-end test case of word count job.

```
class Test(TestCase):

    @classmethod
    def setUp(self):
        print('\r\nSetting up and executing the job')
        self.s3_client = boto3.client('s3', aws_access_key_id='',
            aws_secret_access_key='',
            region_name=StaticVariables.DEFAULT_REGION,
            endpoint_url='http://localhost:4572')
        static_job_info_path = "configuration/static-job-info.json"
        static_job_info_file = open(static_job_info_path, 'r')
        self.static_job_info = json.loads(static_job_info_file.read())
        static_job_info_file.close()
        os.environ["serverless_mapreduce_role"] = 'dummy-role'

    @classmethod
    def tearDown(self):
        print('\r\nTearing down the job')
        del os.environ["serverless_mapreduce_role"]

    def test_that_word_count_returns_correct_results(self):
        # Execute the job
        serverless_mr = ServerlessMR()
        submission_time =
            serverless_mr.map(produce_counts).reduce(aggregate_counts, 4).run()

        print("Job terminated")

        # Retrieve the outputs produced by reducer 2
        job_name = self.static_job_info[StaticVariables.JOB_NAME_FN]
        output_bucket = self.static_job_info[StaticVariables.OUTPUT_SOURCE_FN]
        output_bin = 2
        output_full_prefix = "%s/stage/%s/%s" % (job_name, submission_time,
            str(output_bin))
        response = self.s3_client.get_object(Bucket=output_bucket,
            Key=output_full_prefix)
        contents = response['Body'].read()
        results = json.loads(contents)

        # Assertions
        self.assertIn(['Microsoft', 2], results)
        self.assertIn(['IBM', 2], results)
        self.assertIn(['Software', 1], results)
```

A.4 Extended Min Utilisation's Derivation

Assuming that tasks $T_1 ... T_n$ are to be run with execution frequency ratios of $\lambda_1 ... \lambda_n$ in both frameworks:

$$\text{Min num of executions required in P} = \frac{\text{Cluster cost in P}}{\sum_{i=1}^{n} \lambda_\text{i} * s_\text{i}} \tag{A.1}$$

$$\text{Min Utilisation} = \frac{\text{Min total busy time of the cluster}}{\text{Total time of the cluster}} \tag{A.2}$$

$$= \frac{\text{(Min num of executions in P)} * (\sum_{i=1}^{n} \lambda_\text{i} * e_\text{i})}{\text{Number of seconds in P}} \tag{A.3}$$

$$= \frac{\text{(Cluster cost in P)} * (\sum_{i=1}^{n} \lambda_\text{i} * e_\text{i})}{(\sum_{i=1}^{n} \lambda_\text{i} * s_\text{i}) * \text{(Number of seconds in P)}} \tag{A.4}$$

$$= \frac{\text{(Cluster cost per second)} * (\sum_{i=1}^{n} \lambda_\text{i} * e_\text{i})}{\sum_{i=1}^{n} \lambda_\text{i} * s_\text{i}} \tag{A.5}$$

$$= \frac{\sum_{i=1}^{n} (\lambda_\text{i} * e_\text{i} * \text{(Cluster cost per second)})}{\sum_{i=1}^{n} \lambda_\text{i} * s_\text{i}} \tag{A.6}$$

$$= \frac{\sum_{i=1}^{n} \lambda_\text{i} * h_\text{i}}{\sum_{i=1}^{n} \lambda_\text{i} * s_\text{i}} \tag{A.7}$$

where

P = time period
λ_i = T_i's execution frequency ratio
e_i = T_i's execution time in Hadoop
s_i = T_i's execution cost in ServerlessMR
h_i = T_i's execution cost in Hadoop

Figure A.1: Derivation of extended Minimum Utilisation.

The minimum number of executions required in time period P can also be expressed in terms of the minimum utilisation:

$$\text{Min num of executions in P} = \frac{\text{(Min utilisation)} * \text{(Number of seconds in P)}}{(\sum_{i=1}^{n} \lambda_{\text{i}} * e_{\text{i}})} \tag{A.8}$$

Bibliography

[1] 11 Things You Wish You Knew Before Starting with DynamoDB - The YugaByte Database Blog. Available: https://blog.yugabyte.com/11-things-you-wish-you-knew-before-starting-with-dynamodb/. Accessed on: 2020-06-01.

[2] 2018 Serverless Community Survey: huge growth in serverless usage. Available: https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/. Accessed on: 2020-01-23.

[3] Ad Hoc Big Data Processing Made Simple with Serverless MapReduce — AWS Compute Blog. Available: https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/. Accessed on: 2020-01-23.

[4] Apache Spark Example: Word Count Program in Java - JournalDev. Available: https://www.journaldev.com/20342/apache-spark-example-word-count-program-java. Accessed on: 2020-01-23.

[5] Apache Spark Scheduler. Available: https://databricks.com/session/apache-spark-scheduler. Accessed on: 2020-01-23.

[6] Awslabs/lambda-refarch-mapreduce. Available: https://github.com/awslabs/lambda-refarch-mapreduce. Accessed on: 2020-01-23.

[7] Can I trigger lambda only when message count reaches target value in SQS? - Stack Overflow. Available: https://stackoverflow.com/questions/54089801/can-i-trigger-lambda-only-when-message-count-reaches-target-value-in-sqs. Accessed on: 2020-06-02.

[8] Cloud Data Platform - Big Data Software — Qubole. Available: https://www.qubole.com/. Accessed on: 2020-01-23.

[9] Cloudpipe/cloudpickle: Extended pickling support for Python objects. Available: https://github.com/cloudpipe/cloudpickle. Accessed on: 2020-01-23.

[10] Continuous Integration and Delivery of Apache Spark Applications at Metacog - The Databricks Blog. Available: https://databricks.com/blog/2016/04/06/continuous-integration-and-delivery-of-apache-spark-applications-at-metacog.html. Accessed on: 2020-06-11.

[11] CSRM – Cloud Shared Resource Model – Open Alliance for Cloud Adoption. Available: https://www.oaca-project.org/2018/10/13/csrm-cloud-shared-resource-model/. Accessed on: 2020-01-23.

[12] Data-Intensive Text Processing with MapReduce: Jimmy Lin, Chris Dyer: Amazon.com: Kindle Store. Available: https://books.google.co.uk/books?id=VaddAQAAQBAJ&pg=PA21. Accessed on: 2020-06-02.

[13] Design Patterns using DynamoDB. Available: https://www.slideshare.net/AmazonWebServices/design-patterns-using-amazon-dynamodb. Accessed on: 2020-06-02.

[14] DynamoDB Cheatsheet – Everything you need to know about Amazon Dynamo DB for the 2020 AWS Certified Developer Associate Certification. Available: https://www.freecodecamp.org/news/ultimate-dynamodb-2020-cheatsheet/. Accessed on: 2020-06-01.

[15] DynamoDB Core Components - Amazon DynamoDB. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html#HowItWorks.CoreComponents.PrimaryKey. Accessed on: 2020-06-01.

[16] Google Trend on "serverless". Available: https://trends.google.com/trends/explore?date=2015-01-01%202019-10-01&geo=US&q=serverless. Accessed on: 2020-01-23.

[17] How to Optimize Lambda Memory and CPU - DEV. Available: https://dev.to/byrro/how-to-optimize-lambda-memory-and-cpu-4dj1. Accessed on: 2020-06-07.

[18] Intel-bigdata/HiBench: HiBench is a big data benchmark suite. Available: https://github.com/Intel-bigdata/HiBench. Accessed on: 2020-06-07.

[19] Interface Reducer (Apache Hadoop Main 2.6.0 API). Available: https://hadoop.apache.org/docs/r2.8.0/api/org/apache/hadoop/mapred/Reducer.html. Accessed on: 2020-06-02.

[20] Introducing Corral: A Serverless MapReduce Framework — Ben Congdon. Available: https://benjamincongdon.me/blog/2018/05/02/Introducing-Corral-A-Serverless-MapReduce-Framework/. Accessed on: 2020-05-02.

[21] LeetCode - Course Schedule II. Available: https://leetcode.com/problems/course-schedule-ii/solution/. Accessed on: 2020-06-03.

[22] Localstack: A fully functional local AWS cloud stack. Available: https://github.com/localstack/localstack. Accessed on: 2020-06-02.

[23] MapReduce - Distributed Computing in Java 9 [Book]. Available: https://www.oreilly.com/library/view/distributed-computing-in/9781787126992/5fef6ce5-20d7-4d7c-93eb-7e669d48c2b4.xhtml. Accessed on: 2020-01-23.

[24] MapReduce - When do reduce tasks start in Hadoop? - Stack Overflow. Available: https://stackoverflow.com/questions/11672676/when-do-reduce-tasks-start-in-hadoop/11673808. Accessed on: 2020-06-12.

[25] Qubole offers Apache Spark on AWS Lambda. Available: https://www.qubole.com/blog/spark-on-aws-lambda/. Accessed on: 2020-01-23.

[26] Qubole/spark-on-lambda: Apache Spark on AWS Lambda. Available: https://github.com/qubole/spark-on-lambda. Accessed on: 2020-01-23.

[27] S3 read-after-write consistency for PUTS of new objects. Available: https://medium.com/@kyle.galbraith/the-third-scenario-is-demonstrating-this-line-from-the-aws-documentation-amazon-s3-provides-8b814ce265b3. Accessed on: 2020-06-01.

[28] Spark Execution Flow – experience@imaginea. Available: https://blog.imaginea.com/spark-execution-flow/. Accessed on: 2020-01-23.

[29] Spark Scheduler Backend. Available: https://mallikarjuna_g.gitbooks.io/spark/spark-scheduler-backends.html. Accessed on: 2020-01-23.

[30] Spark/examples at master · apache/spark. Available: https://github.com/apache/spark/tree/master/examples. Accessed on: 2020-01-23.

[31] Top 5 AWS S3 must knows — Cloudten Cloud Consulting. Available: https://www.cloudten.com.au/top-5-aws-s3-must-knows-for-consultants/. Accessed on: 2020-06-01.

[32] Working with Items and Attributes - Amazon DynamoDB. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html#WorkingWithItems.AtomicCounters. Accessed on: 2020-06-01.

[33] AWS - What is cloud computing? Available: https://aws.amazon.com/what-is-cloud-computing/, 2012. Accessed on: 2020-01-23.

[34] Big Data Benchmark. In *Encyclopedia of Big Data Technologies*. 2019, pp. 220–220. Available: https://amplab.cs.berkeley.edu/benchmark/. Accessed on: 2020-06-03.

[35] ADZIC, G., AND CHATLEY, R. Serverless Computing: Economic and Architectural Impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, Association for Computing Machinery, pp. 884–889.

[36] AMAZON. AWS Lambda Limits - AWS Lambda. Available: https://docs.aws.amazon.com/lambda/latest/dg/limits.html, 2014. Accessed on: 2020-01-23.

[37] AMAZON. Amazon DynamoDB - Overview. Available: https://aws.amazon.com/dynamodb/, 2019. Accessed on: 2020-06-03.

[38] AMAZON. Read Consistency - Amazon DynamoDB. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html, 2019. Accessed on: 2020-06-01.

[39] AMAZON. Amazon EMR - Big Data Platform - Amazon Web Services. Available: https://aws.amazon.com/emr/faqs/, 2020. Accessed on: 2020-06-07.

[40] AMAZON. S3 Simple Storage Service pricing - Amazon Web Services. Available: https://aws.amazon.com/s3/pricing/, 2020. Accessed on: 2020-06-06.

[41] AMAZON WEB SERVICES. AWS Lambda Function Handler in Python. Available: https://docs.aws.amazon.com/lambda/latest/dg/python-handler.html. Accessed on: 2020-06-10.

[42] AMAZON WEB SERVICES. Serverless Computing – Amazon Web Services. Available: https://aws.amazon.com/serverless/, 2017. Accessed on: 2020-01-23.

[43] AMAZON WEB SERVICES INC. Introduction to Amazon S3 - Amazon Simple Storage Service. Available: https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel, 2018. Accessed on: 2020-06-01.

[44] ANTON KIRILLOV. Apache Spark: core concepts, architecture and internals. Available: http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/, 2016. Accessed on: 2020-01-23.

[45] AWS. AWS Lambda – Serverless Compute. Available: https://aws.amazon.com/lambda/, 2018. Accessed on: 2020-01-23.

[46] AWS COMPUTE BLOG. Understanding Container Reuse in AWS Lambda — AWS Compute Blog. Available: https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/. Accessed on: 2020-01-23.

[47] BALDINI, I., CASTRO, P., CHANG, K., CHENG, P., FINK, S., ISHAKIAN, V., MITCHELL, N., MUTHUSAMY, V., RABBAH, R., SLOMINSKI, A., AND SUTER, P. *Serverless Computing: Current Trends and Open Problems*. Springer Singapore, Singapore, 2017, pp. 1–20.

[48] BARR, J. New – Per-Second Billing for EC2 Instances and EBS Volumes — AWS News Blog. Available: https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/, 2017. Accessed on: 2020-06-07.

[49] CHAPIN, J. The Occasional Chaos of AWS Lambda Runtime Performance. Available: https://blog.symphonia.io/the-occasional-chaos-of-aws-lambda-runtime-performance-880773620a7e, 2017. Accessed on: 2020-06-03.

[50] CREATIVE TIM. Light Bootstrap Dashboard: Free Bootstrap Admin Template @ Creative Tim. Available: https://www.creative-tim.com/product/light-bootstrap-dashboard, 2015. Accessed on: 2020-06-03.

[51] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (San Francisco, CA, 2004), pp. 137–150.

[52] DEYHIM, P. Best Practices for Amazon EMR. No. August. 2013, pp. 1–38. Available: `https://d0.awsstatic.com/whitepapers/aws-amazon-emr-best-practices.pdf`, page: 14.

[53] ESLAM HEFNAWY. Serverless Code Patterns. Available: `https://www.serverless.com/blog/serverless-architecture-code-patterns/`, 2019. Accessed on: 2020-06-03.

[54] FOULADI, S., ROMERO, F., ITER, D., LI, Q., CHATTERJEE, S., KOZYRAKIS, C., ZAHARIA, M., AND WINSTEIN, K. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)* (Renton, WA, jul 2019), {USENIX} Association, pp. 475–488.

[55] FOUNDATION, T. A. S. MapReduce Tutorial. *Source* (2008), 1–43. Available: `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`. Accessed on: 2020-01-23.

[56] GARCÍA-LÓPEZ, P., SÁNCHEZ-ARTIGAS, M., SHILLAKER, S., PIETZUCH, P., BREITGAND, D., VERNIK, G., SUTRA, P., TARRANT, T., AND FERRER, A. J. ServerMix: Tradeoffs and Challenges of Serverless Data Analytics, 2019.

[57] GURU99. Hadoop & Mapreduce Examples: Create your First Program. Available: `https://www.guru99.com/create-your-first-hadoop-program.html`. Accessed on: 2020-01-23.

[58] HELLERSTEIN, J. M., FALEIRO, J., GONZALEZ, J. E., SCHLEIER-SMITH, J., SREEKANTI, V., TUMANOV, A., AND WU, C. Serverless Computing: One Step Forward, Two Steps Back, 2018.

[59] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the Cloud: Distributed Computing for the 99%, 2017.

[60] KIM, Y., AND LIN, J. Serverless Data Analytics with Flint. In *IEEE International Conference on Cloud Computing, CLOUD* (2018), vol. 2018-July, pp. 451–455.

[61] KIM, Y., AND LIN, J. Serverless Data Analytics with Flint (Thesis), 2018. Available: `http://hdl.handle.net/10012/13681`.

[62] KOLB, R. W. Apache Hadoop. Available: `https://hadoop.apache.org/`, 2018. Accessed on: 2020-01-23.

[63] MALISHEV, N. AWS Lambda Cold Start Language Comparisons, 2019 edition. Available: `https://levelup.gitconnected.com/aws-lambda-cold-start-language-comparisons-2019-edition-%EF%B8%8F-1946d32a0244`. Accessed on: 2020-06-01.

[64] MAPR. Apache Hive – What It Is, What It Does, and Why It Matters? Available: https://mapr.com/products/apache-hadoop/, 2019. Accessed on: 2020-01-23.

[65] MICROSOFT CORPORATE. What is cloud computing? A beginner's guide — Microsoft Azure. Available: https://azure.microsoft.com/en-gb/overview/what-is-cloud-computing/, 2016. Accessed on: 2020-01-23.

[66] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2009), SIGMOD '09, Association for Computing Machinery, pp. 165–178.

[67] PIETZUCH, P. Lecture Materials of CO410 Scalable Systems and Data.

[68] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (USA, 2019), NSDI'19, USENIX Association, pp. 193–206.

[69] THE PYTHON SOFTWARE FOUNDATION. pickle — Python object serialization — Python 3.7.3 documentation. Available: https://docs.python.org/3/library/pickle.html, 2019. Accessed on: 2020-06-01.

[70] TUTORIALKART. Understand DAG and Physical Execution Plan in Apache Spark. Available: https://www.tutorialkart.com/apache-spark/dag-and-physical-execution-plan/. Accessed on: 2020-01-23.

[71] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (San Jose, CA, 2012), USENIX, pp. 15–28.