CS

21COC251

B713109

Ribble: A serverless data processing framework

by

Jose Narvaez Paliza

Supervisor: Dr. L Nagel

Department of Computer Science

Loughborough University

May 2022

Abstract

Data processing frameworks like Apache Hadoop and Apache Flink have been popularized given their power to process large amounts of data in parallel while abstracting issues that distributed systems naturally have. These frameworks offer different features and programming models; however, they share limitations: cluster management can become a bottleneck, low-utilized clusters may incur high costs, and they can have a steep learning curve for inexperienced programmers. In this dissertation, we present Ribble, a serverless framework for distributed data processing. We provide an easy-to-use Go interface that allows data analysts and programmers to process data in parallel. Ribble removes cluster management and configurations and allows users to exploit the scalability and pay-as-you-go billing model of serverless computing. To achieve this, we focus on delivering a framework that transparently handles fault-tolerance, data distribution, and scalability. We explore and evaluate a framework based on a data-flow engine using data streams as its data distribution mechanism.

Table of contents

1 Introduction

Writing code that can be parallelized is difficult even for experienced programmers. While many programming languages offer a good range of primitives that can help writing parallel code, programmers are used to writing code that runs sequentially. Additionally, distributed applications introduce problems that are usually not considered when writing sequential code. To name a few, programmers need to consider mechanisms to achieve synchronization, consistency, fault-tolerance, and scalability.

Many frameworks and programming libraries have been implemented to allow programmers to write parallel programs more easily. Frameworks like MapReduce allows users to define data processing jobs where the data distribution, scheduling, and fault tolerance is handled by the framework and made transparent to the users. Other approaches like the Message Parsing Interface (MPI) offer a standard that allows users to define a message exchange mechanism between computers that run on parallel computing architectures. While these approaches facilitate implementing parallel workloads, users still need to know how the distribution works to write the code.

The motivation of this project is to provide a framework that can automatically parallelize sequential code written by the users. While it would be interesting and extremely useful to achieve this for general purpose computations, it would be outside the scope for this project because of its difficulty and time constraints. Instead, this project focuses on developing a framework that allows users to write sequential code that can be parallelized to process big data.

Big data refers to structure and unstructured data that cannot be placed in traditional relational databases because of its massive size. Because of improvements of data storage technologies, companies have opted to capture large amounts of data. If the data is properly analyzed, it has the potential to transform businesses by gaining a competitive advantage. However, with the size of the data, existing tools such as SQL and other data mining technologies cannot analyze data efficiently. For this reason, distributed processing frameworks are essential. Additionally, data mining applications are usually run by analysts who often have no experience in parallel computing, reason why a framework that abstracts the parallelization logic and transparently handles the distributed mechanisms is necessary.

Parallelization is achieved when the computation or data can be distributed across many servers and run at the same time. This implies that analysts that want to run distributed data processing heavily rely on the infrastructure they have available. Not long ago, users were restricted to their on-premises infrastructure, however the rise of cloud computing has enabled users to deploy workloads in infrastructure managed by external providers. Early delivery models like infrastructure-as-a-service (IaaS) allowed users to rent infrastructure usually in the form of virtual machines (VMs). While it provides users with elasticity, the setup and initialization times of the servers do not allow users to scale up and down servers efficiently to meet the demand of the applications and they usually spend money on resources that were never used.

Serverless computing is a paradigm that removes server management completely. It allows users to run their workloads on an infrastructure that is highly scalable and offers a fine-grained billing model where users pay exactly for what they use. For ad-hoc queries such a paradigm seems ideal, however,

serverless function have restricted network connectivity, limited run time, volatile storage, and there is no control over the servers used to run the functions. These factors make a serverless data processing framework difficult to implement.

In this project we focus on developing a data processing framework that:

1. Offers a programming model that allows users to sequentially define data processing jobs that can be parallelized
2. Can scale to meet the demands of big data
3. Makes the distributed mechanisms such as parallelization, scalability, synchronization, and fault-tolerance mechanisms transparent
4. Allows for easy and secure integration with the user's cloud resources
5. Runs as fully serverless framework
6. Offers a pay-as-you-go billing model
7. Provides users with a utility to run and track jobs
8. Provides users with a local environment where they can test jobs
9. Provides a framework that has been tested and evaluated

2 Background

2.1 Data processing frameworks

2.1.1 Apache Hadoop

Apache Hadoop is a distributed data processing framework that runs on clusters of commodity servers to process massive amounts of data in batch. Batch datasets are bounded, persistent, and large, reason why Apache Hadoop is used to process large and static historical data when the processing speed is not critical to the user. Hadoop is composed of three core elements: The Hadoop Distributed File System (HDFS), YARN (Yet Another Resource Negotiator) and the MapReduce engine.

The Hadoop Distributed File System (HDFS)

HDFS is a distributed file system that coordinates storage and replication across the cluster. It provides a reliable, scalable and fault tolerant storage layer that runs on top of unreliable commodity servers.

YARN (Yet Another Resource Negotiator)

MapReduce relies on a master-worker architecture. The master node is responsible for scheduling map and reduce tasks to idle workers and subsequently, each worker is responsible for executing a map or reduce task. The scheduling mechanism and cluster management are performed by YARN. YARN is composed of three system daemons, the Global Resource Manager (GM), Application Master (AM), and Node Managers (NM). The GM and AM both run in the master node while the NM runs in each of the workers. The GM arbitrates all cluster resources across all running jobs, the AM negotiates resources with the GM and works with the NM to monitor, start, and stop tasks (Glushkova et al., 2019).

The MapReduce engine

MapReduce is a programming model and implementation used to process large datasets of raw, non-structured data. MapReduce allows users with no previous knowledge of distributed systems to run data processing jobs in a cluster of computers. This is possible given that programs that can be expressed with MapReduce are automatically parallelizable and its implementation hides faut-tolerance, parallel processing, data-distribution, load-balancing, and scalability of the system.

MapReduce consists of two main stages: map and reduce. Both stages need user-defined functions. Map processes the input data and produces a set of intermediate key-value pairs and reduce aggregates all intermediate key-value pairs that share the same key (Dean et al., 2004).

Workflow

1) Job initialization: A MapReduce job is first initialized by the user, and it must at least specify the dataset to process and the map and reduce functions to use.

2) Data splits and mapper invocations: MapReduce distributes the map invocations by automatically partitioning the input dataset into multiple splits (usually 64MB). Each of these splits is independent of each other meaning that they can all be processed in parallel if enough servers are available. Once the splits are produced, the master node assigns each idle worker a data split to work on.

3) Processing input splits and data shuffling: Each worker node reads its corresponding input split and processes the data according to the user-defined map function. The output of each node is written back to the distributed file system using a data shuffling mechanism where each key-value pair is written into a specific partition given by a hash-partition function.

4) Reducing intermediate key-value pairs: Once the workers are done with the map stage, each reducer is notified with the location of the intermediate data it is responsible for. There is a one-to-one mapping from partitions to reducers, meaning that each partition will be processed by a single reducer. Reducers read the data from the distributed file system using remote procedure calls (RPCs) and sort the data alphabetically by keys. Once the data is sorted, it is grouped by key and each grouping is processed together using the user-defined reduce function. Once the Reducer processes all data in its partition, its output is written back to the distributed file system.

5) Job output: The job ends when all reducers have completed their processing. The output of the job is available in the output files of the reducers. If there are N partitions the output will be split into N different output files, and these can be merged or used as the input for another MapReduce job.

Limitations

The MapReduce framework does make it easier for programmers to write data processing jobs given that the framework provides a restricted programming model and provides a transparent fault-tolerance system. However, to write more complex jobs, where additional extensions are needed such as writing a user defined partition function or overriding the grouping functionality requires users to have an in-depth knowledge of MapReduce. Analysts writing data processing jobs may not be experienced programmers, which will make it difficult to use.

2.1.2 Apache Flink

Apache Flink is another Apache open-source framework for distributed data processing. Like Apache Hadoop, its aim is to facilitate distributed data processing by abstracting data distribution, communication, and fault tolerance to the users. The main difference to Apache Hadoop is that its core uses a distributed data-flow engine that uses data streams as its mechanism to distribute data (Bergamaschi et al., 2017). This allows the framework to perform batch and real-time stream processing. In this section, we will mainly focus on how Flink supports batch processing on top of a streaming engine.

Flink applications are streaming dataflows represented as direct acyclic graphs (DAG) composed of a data source, operators, streams, and a sink. Operators are transformations that can be applied to an input stream and the output of these transformations is sent to an output stream which can serve as the input for the next set of operators. The source is the entry point for streams entering the system and the sink is the component where output data is written too, for example, a distributed database such as HDFS. Figure 1 shows a DAG that represents a Flink job, we can see *map()* as the first operator transforming the source data and a set of operators k*eyBy()*, *window()* and *apply()* transforming the data stream produced by *map().*

Distributed data processing is achieved by using stream partitions and operator subtasks. Operator subtasks are independent from each other, and each subtask works on a different stream partition. This

level of isolation means that each operator subtask can be performed in a different thread or machine, thus allowing the system to increase its parallelization factor.

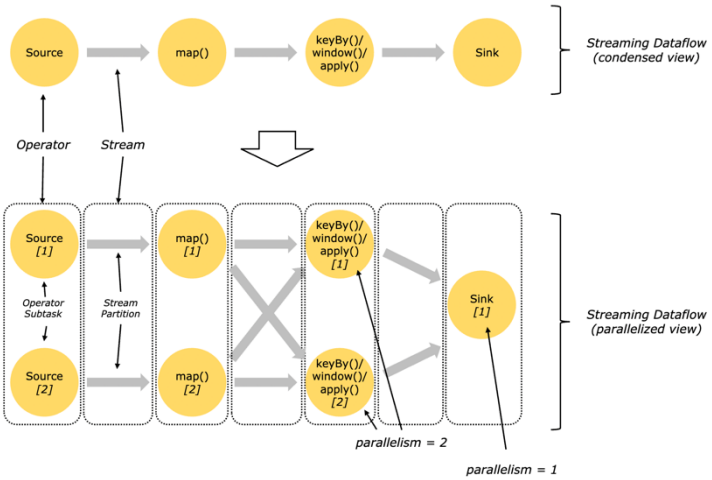


Figure 1: Flink's parallel dataflow (https://nightlies.apache.org/flink/flink-docs-release-1.14/fig/learn-flink/parallel_dataflow.svg, consulted on 03/01/2022)

Architecture and execution flow

Flink relies on a master-worker architecture. The master or JobManager, coordinates the execution of the job by tracking the status of operators and streams, scheduling new operators and orchestrating checkpoints that are used as a fault-tolerant mechanism. The worker nodes or TaskManagers execute operators that produce streams and track their progress. Apart from the master and workers, Flink uses a Job Client which is responsible for transforming the user defined program into a Flink data-flow representation and submitting the job to the JobManager (https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/concepts/flink-architecture/, consulted on 03/01/2022).

A Flink job goes through the following steps:

1) The Flink program is first submitted to the Job Client. The Job Client converts the Flink program into a dataflow graph and submits it to the JobManager. Additionally, the program goes through an optimization phase.

2) The JobManager receives the dataflow program and coordinates its execution by allocating operators to the available TaskManagers. Apart from scheduling and tracking the operator and streams, the JobManager continuously performs a checkpointing mechanism that consists of saving a minimal set of metadata to persistent storage.

3) TaskManagers report to the JobManager about their status and execute the operators.

4) The execution of the dataflow program ends when a sink operator is reached.

Stateful operations and fault tolerance

Some Flink programs require operators to keep their state to remember operations across multiple states. A simple example of a stateful operation is computing an average where the TaskManager needs to keep track of the previous messages to compute the correct result. In case of a TaskManager failure,

Flink uses re-execution and checkpointing as its fault-tolerant strategy. Instead of replaying all messages from a stream, Flink uses checkpoints to save the current state of the operator and the current point on the stream. A failed TaskManager can execute from where the failing worker stopped hence achieving exactly once processing (https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/concepts/stateful-stream-processing/, consulted on 03/01/2022).

Checkpointing is achieved using stream barriers. Barriers are introduced at intervals into the streams and flow alongside the messages of the stream. The position of the barriers represents the set of records that are included in each checkpoint. This allows TaskManagers that restore from a saved checkpoint to read messages that are after the last barrier.

For batch processing, checkpointing can be disabled given the bounded nature of streams. To recover from a failure in this case, the TaskManager needs to re-play the entire stream again. For this reason, streams need to be re-playable. This setup allows the system to perform quicker in case of no failures, however, it will take longer to recover from a failure.

DataSetAPI

Instead of using MapReduce as its programming model, Flink consists of an API that users use to write programs that are later translated into parallel distributed computations. For batch processing, Flink uses DataSet API, a dedicated API that provides abstractions for batch computations such as filtering, mapping, joining, and grouping (Carbone et al., 2016).

2.1.3 Sawzall

Sawzall is a procedural programming language developed by Google and is used to process large datasets that often do not fit in traditional relational databases. Sawzall processes data stored on GFS (Google File System) and runs on top of MapReduce which allows it to execute operations for different data items in parallel while abstracting its distributed aspects such as scheduling and fault tolerance.

Sawzall offers less expressiveness than SQL queries, but it reiterates many data analytic applications do not need the sophistication that SQL queries provide. While it recognizes procedural programming languages can achieve the same as Sawzall, languages like Python do not have the facility to distribute computation to thousands of computers without adding a burden to programmers. Additionally, Sawzall assures that compared to C++ code running MapReduce, it can be much simpler and shorter, up to 10 times (Pike et al., 2005).

Sawzall's main advantage over other distributed data processing frameworks is that it simplifies the writing of programs. This is achieved by forcing the programmers to write code thinking on how to process a single data item at a time and by not requiring users to write the aggregators but instead offers a set of already implemented aggregators users can choose from. Programmers that use Sawzall can write programs that will process data in parallel without knowing anything about distributed systems. However, this comes with its disadvantages. First, programmers need to learn yet another programming language. Second, the expressive power of the language is restricted to the available aggregators. And third there is no concept of chaining jobs which is commonly included in data processing frameworks.

Data analytics in Sawzall is comprised of a query and an aggregation stage, where the query runs in the map phase of MapReduce and the aggregation in the reduce phase. Sawzall focuses on commutative and associative query and aggregation operations which allows it to process data and group intermediate values in any order. Some of the aggregators used in Sawzall are the following:

- Collection: offers a list of emitted values
- Sample: offers a sample from a collection of emitted values
- Sum: adds up the emitted values
- Maximum: returns the maximum value emitted
- Top: returns the most frequent values emitted

To write a program in Sawzall, programmers need to specify 1) the aggregators, also referred as tables, 2) how the input is processed and 3) what values are emitted to the aggregators.

```
count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;
```

Figure 2: Example of a Sawzall program (Pike et al., 2005)

Figure 2 shows an example of a Sawzall program where it reads input and produces the number of records, the sum of the values, and the sum of the square. The execution of such a program is not complicated. First a job request is received by the system, after validating the Sawzall source code, it is sent to the machines for execution. These machines compile the Sawzall source code and run it to process individual data items. The values emitted are then aggregated locally, like the combine phase in MapReduce, before sending them to the corresponding machines. The aggregator machines collect the intermediate output and aggregate the values accordingly and sort the output before writing it back to GFS. At the end of the job there is a file in GFS for each aggregator machine.

2.2 Serverless Computing

In general, serverless computing can be divided into two categories: Functions-as-a-service (FaaS) and Backend-as-a-service (BaaS) offerings. Completely serverless systems rely on both, although the introduction of FaaS offerings has popularized the term serverless computing.

Serverless services are characterized by:

1) No explicit management of servers: specifically for cloud functions, users write code in high-level languages and the service provider provisions and administrates the servers. Specifically, the provider manages instance selection, scaling, deployment, fault tolerance, monitoring, logging and security patches (Jonas et al., 2019). This is similar with services such as object storage and distributed queue solutions where the user is not aware of data distribution, replication, fault-tolerance and scalability.

2) Pay-as-you-go billing model: users of serverless services pay exactly for that they use and there is a finer grain billing system.

In the following section we discuss serverless services provided by Amazon Web Services (AWS). In particular, we discuss AWS Lambda as a FaaS offering and AWS S3 and AWS SQS as BaaS offerings.

2.2.1 Lambda

AWS Lambda (Lambda) is the FaaS offering from AWS. Lambda is divided into two main components, the lambda service and lambda functions. The lambda service is responsible for providing a high-availability compute infrastructure for the lambda functions and manages provisioning, scaling, monitoring among other functionalities provided by the service. The lambda functions are the fundamental principle, they are the unit of computations of the service.

The programming model provided by Lambda offers an interface between the user's code and the service. Users must define a handler as shown below, which is used as the entry point of the function, and it is used to define how new events should be processed. The following code snippet illustrates an implementation written in Go of the handle function which process events of type MyEvent. Note that the handle receives a context object and an event. The context provides methods and information about the function's environment. The event is a JSON formatted object containing data for the function to act upon.

```
func HandleRequest(ctx context.Context, name MyEvent) (string, error) {
	return fmt.Sprintf("Hello %s!", name.Name ), nil
}
```

Figure 3: example of a function handler (https://docs.aws.amazon.com/lambda/latest/dg/golang-handler.html, consulted on 1/5/2022)

Because Lambda functions need to be defined as executables, a main function needs to be implemented and serves as the entrypoint of a function. As shown below, the Lambda package offered by the AWS SDK for Go can be used to define it:

```
func main() {
	lambda.Start(HandleRequest)
}
```

Figure 4: example of a function entrypoint (https://docs.aws.amazon.com/lambda/latest/dg/golang-handler.html, consulted on 1/5/2022)

With this definition, Lambdas can be created and every time there is a new event in the Lambda service, an instance of the function is spined up to process it.

Features

1) Scaling and concurrency

The Lambda service scales automatically in response to invocations. The amount of lambda functions running at the same time is referred to as the concurrency of a lambda function. The concurrency of a function increases depending on the demand and the available functions. When a function is invoked faster than it can process new events, new function instances are provisioned. When this happens, the

Lambda service needs to initialize the function before it can process events. This is called a cold start and this start-up time can increase latency significantly. To deal with cold starts, AWS allows functions to be configured with provisioned concurrency, which initializes instances before time to make sure functions are ready to receive events.

With highly parallelizable workloads it would be ideal to provide as many functions as possible such that all processing can be done in parallel. However, this is not possible since AWS limits the level of concurrency to protect resources that are shared across all customers using the service. By default, there is a concurrency limit of 1,000. Apart from this overall limit, there is a limit to the number of functions that can be initialized at once, called burst concurrency.

2) Fault-tolerance using asynchronous invocation

For invoking functions asynchronously, invocations are sent as events to the Lambda service. Lambda then takes events from this queue and invokes the functions if the concurrency limit has not been reached. This queuing mechanism allows the function to have a natural way to handle errors. If a function fails on execution, the function is re-queued hoping that on following retries the function will succeed. The size of the queue allows the service to adjust the invocation rate to reduce the number of functions that fail due to a concurrency limit error.

Limitations

Resource	Quota
Function memory allocation	128 MB to 10,240 MB, in 1-MB increments.
Function Timeout	15 minutes
Concurrency Limit	1,000 (can be increased to thousands)
Burst Concurrency	500 - 3000 (varies per region)
Invocation payload	256 KB (asynchronous)
Local file system storage	512 MB

Table 1: AWS Lambda limits (https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html accessed on 05/01/2022)

2.2.2 SQS

Amazon Simple Queue Service (SQS) is one of the first AWS services to be offered as a serverless BaaS with a pay-as-you go billing model. SQS provides a simple queue system that allows distributed applications to share data. It has gained popularity for such systems as the service abstracts redundancy, durability, high-availability, and automatic scalability of queues.

SQS provides two types of queues: standard and FIFO. As its name suggests, FIFO provides a first-in-first-out delivery mechanism, and it guarantees exactly-once processing. Given the low throughput FIFO queues allow, we consider standard queues a better fit for data intensive applications. In the other hand, standard queues guarantee at-least once delivery and best-effort ordering. This means that systems working with standard queues will need to consider implementing idempotent processing mechanisms and may not rely on messages to be received in the same order as they were sent in. However, this cost comes with a great reward: almost unlimited throughput.

Limitations

Resource	Quota
Messages per queue (backlog)	Unlimited
Messages per queue (in flight)	120,000 inflight messages
Message throughput	Unlimited
Message size	262,144 bytes (256 KB)

Table 2: AWS SQS limits
(https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dg.pdf, accessed on 05/01/2022)

2.2.3 S3

Amazon Simple Storage Service (S3) is a scalable object storage service that allows users with varied use cases to store and protect their data using a pay-as-you go billing model.

S3 main components are buckets and objects. Objects are files along with their metadata and buckets contain objects, hence an object is identified by the bucket and object key. While data in buckets can be organized to imitate the structure of a filesystem using prefixes, it is important to note that there is a distinction between both.

Limitations

1) Throughput: AWS supports 3,500 PUT/COPY/POST/DELETE or 5,500 GET/HEAD requests per second per prefix. Given that there are no restrictions on the number of prefixes you can have, S3 can massively scale if configured correctly.

3) Number of buckets: by default, S3 can only have 100 buckets but can be increased to 1,000. Users cannot rely on using thousands of buckets to structure their applications. Instead, care should be taken to structure the buckets using prefixes with object keys.

4) Object size: An object can be up to 5 TB but it can only be uploaded with a multi-part upload using an the provided S3 SDK or directly though the S3 rest interface.

2.3 Data processing frameworks in serverless architectures

Distributed data processing frameworks provide data analysts with powerful tools to process data in an efficient manner. We have already reviewed popular solutions such as Apache Hadoop and Apache Flink, frameworks that can run in clusters of commodity servers. While running these frameworks in a cluster of servers is the reason behind the distribution and parallelization of data processing, the management of such clusters can become a bottleneck for data analyst or scientists who do not have a strong background on system administration. For instance, figure 5 shows the issues to be addressed in setting up environments in the cloud. Cloud providers have recognized this problematic and have introduced managed clusters such as Amazon Elastic MapReduce (AWS EMR) and Google Cloud Dataproc. While managed clusters are easier to configure, there is still a need for management and configuration, and they suffer from a high initialization time. Additionally, such services are billed using a per-second model, so users with unutilized resources will have a high idle cost (Werner et al., 2020).

1. Redundancy for availability, so that a single machine failure doesn't take down the service.
2. Geographic distribution of redundant copies to preserve the service in case of disaster.
3. Load balancing and request routing to efficiently utilize resources.
4. Autoscaling in response to changes in load to scale up or down the system.
5. Monitoring to make sure the service is still running well.
6. Logging to record messages needed for debugging or performance tuning.
7. System upgrades, including security patching.
8. Migration to new instances as they become available.

Figure 5: Eight issues to be addressed in setting up an environment in the cloud (Jonas et al., 2019)

In this section we evaluate data processing frameworks that run in serverless architectures. In general, these frameworks promise to reduce configuration setup to a minimum, to provide a pay-per-use billing model hence lowering the cost for ad-hoc processing and to compete in terms of performance against more traditional frameworks running on managed clusters. According to an evaluation of serverless functions (Werner et al., 2020), all serverless frameworks outperform an Apache Spark cluster in terms of performance for ad-hoc queries.

Given the popularity of MapReduce, most serverless frameworks discussed are MapReduce inspired serverless frameworks. Overall, most share similar characteristics in terms of their architecture and execution flow. For instance, all frameworks use AWS as the cloud provider, using a combination of Lambda, S3, DynamoDB, and SQS as their core services. Most consist of a combination of a local driver or coordinator Lambda function to orchestrate the execution flow. Regardless of the programming model, processing functions are presented as Lambda functions. In terms of fault tolerance, they use re-execution as their main mechanism and sometimes checkpointing to aid this. The struggle to shuffle data between producers and consumers is something they all share. This is due to the nature of serverless services that have restricted memory and execution time.

Given the similitude between these frameworks, we do not discuss in detail the execution flow and architecture of each, instead we focus on how they achieve fault-tolerance, data shuffling and scalability mechanisms.

2.3.1 Marla

Marla is a serverless framework used to run MapReduce jobs on AWS lambda and S3.

Architecture and execution flow

Figure 6 shows the architecture and execution flow of Marla which is composed of three Lambda functions: a coordinator, mappers, and reducers. Additionally, it uses S3 as the storage connector for the input, output, and intermediate data. Marla execution flow is presented below as a series of steps:

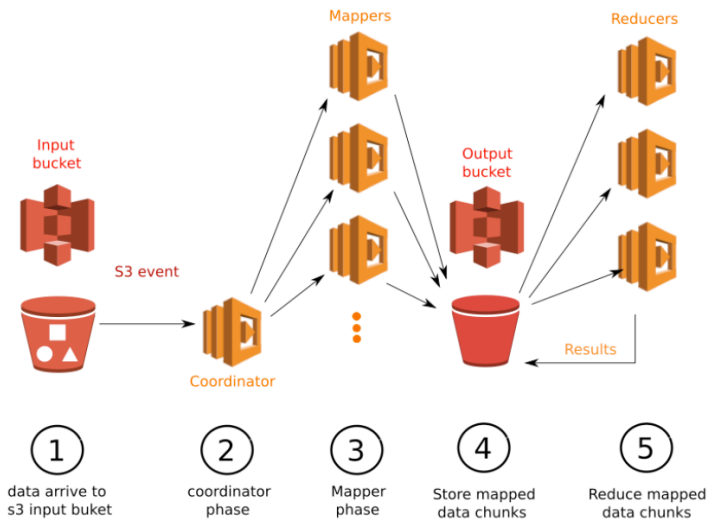


Figure 6: MARLA's execution flow (Giménez-Alventosa et al., 2019)

Step 1: Before execution, the user defines the map and reduce functions using the Python API provided by the framework. These functions are bound to an S3 bucket, such that when files are uploaded to the bucket, it triggers the coordinator function which begins the execution of the job. Note that this allows the framework to be triggered on demand without the need to pre-deploy the service.

Step 2: The coordinator function first responsibility is to split the input data. This split is done automatically by considering the number of mappers the user specified and the amount of memory each Lambda function is configured to use. Once the coordinator has defined the optimal splits it invokes the first lambda mapper. The first mapper, initialized by the coordinator, recursively invokes the rest of the mappers such that each new initialized mapper invokes a new one.

Step 3: Once all mappers are up and running, they act on their respective data chunk and process the data according to the user defined mapper function. After a mapper is finished, its result is stored in S3. The output in S3 will be sorted and split into different partitions grouped by the keys, where each partition has a different S3 key.

Step 4: The reduce phase begins as soon as the last mapper starts. The first reducer waits until the last partition is written to S3 and then triggers the invocation of the rest of the reducers. Using the user defined reduce function, each reducer will process a partition until all partitions have been processed.

Step 5: Finally, once reducers finish up processing all partitions, their output data will be written back to S3 as a set of N S3 keys where N is the number of partitions.

Fault tolerance

Overall, Marla does not provide an extensive mechanism to achieve fault-tolerance. All three main functions rely on re-execution as its fault-tolerant mechanism, however, if the re-execution limit is reached for any of the functions the job will fail. If the coordinator fails, the framework fails to start and will need to be triggered again and no additional resources are created. If the mapper fails, the reducer for that partition will never succeed as the data will not be available. A limitation of the mapper function

is that it does not skip files, which means that a single invalid value in the input data set will make the whole job fail.

Limitations and advantages

Limitations: Marla uses the first letter of the keys as the default partitioning mechanism. If keys are not evenly distributed, this could cause partitions to have very different sizes. Note that the partition function can be overridden by a hash partition function for example, but this forces the users to understand how a partitioning function must be implemented.

There is no coordinator for the reduce step, which means that users will only be aware a job has finished by manually getting the output results from S3.

Advantages: Using recursive invocation of Lambda functions reduces the time needed to invoke functions and it could reduce concurrency limit errors as the invocation of new functions will stop once the first error is reached.

2.3.2 AWS Serverless MapReduce

The simplest of frameworks is the one provided by AWS, nevertheless, AWS states that the solution is cheaper and faster than existing well-known MapReduce frameworks (https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/, consulted on 20/12/2021). This framework allows the users to run ad-hoc MapReduce jobs using a serverless architecture composed of Lambda and S3 as its main components.

This framework is composed of a local driver and three main Lambda functions: the mapper, reducer, and coordinator. The driver is used to initialize the job by splitting the input data into batches and invoking the mappers. Once the mappers have been invoked, the driver is no longer needed, and for that reason we consider this framework to be fully serverless.

Each mapper will process the batch it is assigned and write its output back into S3. Given the absence of a shuffle and sort stage, this implementation will not be able to scale as the input data size increases. This is true as reducers will need to reduce all files recursively until there is one left to achieve a global aggregation. Apart from speed slowdown, we can also see that the framework is heavily restricted by the memory of the reducer Lambda function because the output needs to fit in the memory of the last reducer.

As an interesting takeaway from the execution flow, we discuss the mechanism used to invoke the coordinator. This implementation uses an event-source mapping on the S3 buckets where mapper output is written to. This allows the coordinator to be invoked without the need to have a daemon-like process tracking the status of mappers, instead when an object is written to S3, the coordinator function will be invoked, it will check if the mappers have finished and then start the reduce phase.

2.3.3 Corral

Corral is a MapReduce inspired framework written in Go and designed to run with AWS Lambda. Corral offers easy to use Go interfaces that users can use to define map and reduce functions. It uses S3 as its data distribution mechanism between mappers and reducers.

Architecture

Corral uses a local driver and map and reduce lambda functions. Corral states that the framework is completely serverless (https://benjamincongdon.me/blog/2018/05/02/Introducing-Corral-A-Serverless-MapReduce-Framework/, consulted on 22/12/2021) , however the driver is needed across the entire duration of the job to schedule the mappers and reducers, hence making the framework semi-serverless.

Data shuffling

Like Marla, Corral includes a shuffle stage without sorting. This is shown in figure 7, where mappers write their output into different bins represented by S3 buckets and reducers read each partition to reduce it. In more detail, the keys in the mappers are partitioned into N buckets where N is the number of reducers the framework is configured with. Each mapper writes its output to each of these buckets, for example, we can see that mapper 1 splits its output to bin 1 and bin 3 (nothing is written to bin 2 because the mapper did not have keys that were mapped to that partition). Once mappers are done, each reducer will act on a single bucket. Given that the output is not sorted, a reducer will need to retrieve all objects in the bucket to reduce them correctly. This is a limitation of the framework given that if not correctly configured, the data objects in a respective bucket could be larger than the reducer's memory. While a solution to this would be to increase the number of reducers such that there is less data to process for each one, we encounter two problematics. First, keys are partitioned into different buckets and according to AWS an account can have a maximum of 100 buckets, hence the number of reducers is limited to 99 reducers (considering one is needed for the input bucket). Second, keys are not always correctly distributed, so while increasing the number of reducers will create smaller buckets, we could have a bucket much larger than others.

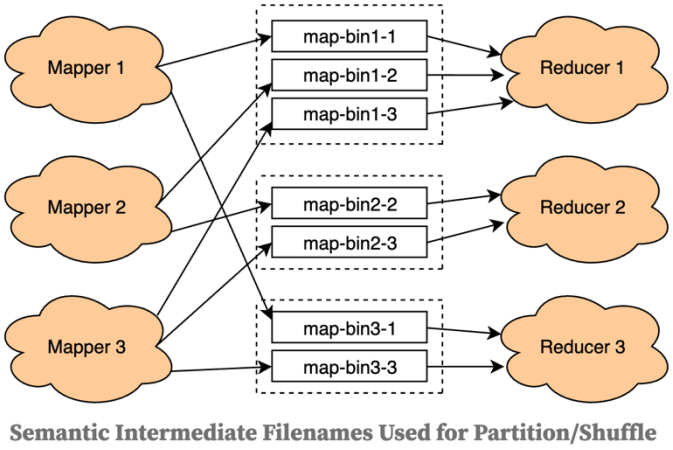


Figure 7: Corral's shuffle mechanism (https://benjamincongdon.me/blog/2018/05/02/Introducing-Corral-A-Serverless-MapReduce-Framework/, consulted on 22/12/2021)

2.3.4 PyWren

PyWren is a framework with a Python API that allows programmers to define embarrassingly parallelizable jobs. This framework provides three execution modes.

1. Map-only: a map abstraction for many scientific and analytic jobs that are embarrassingly parallel.
2. Map and monolithic reduce: this mode introduces a reduce phase where the aggregated values are collected in a single reducer. This is mode is useful with compute intensive applications that do not generate many features. The reducer in this mode does not usually run in a serverless function but in VM.
3. Map and reduce: this mode allows for more general-purpose computations. PyWren states that it can handle shuffle-intensive workloads that can sort 1TB of data in 3.4 minutes (Jonas et al., 2017).

Given its good performance with shuffle intensive workloads we evaluate PyWren's shuffle mechanism in more detail.

Data shuffling

Data shuffling is split into two stages. The first stage partitions the intermediate data generated by the mappers and writes each into S3. The second stage merges and sorts each partition. After this the reducers can aggregate the data from the partitions similar to how MapReduce engines do it.

To achieve the shuffle of 1TB of data, PyWren states that it needs to shuffle around 6 million intermediate files in between. While S3 has a high I/O throughput, it falls short, so it utilizes a Redis cluster to manage the intermediate output. There are two limitations with this solution. First while the Redis cluster (if big enough) could shuffle data faster it eventually becomes the bottleneck of the system. Second, having a Redis cluster means this framework is not completely serverless and it moves away from a pay-per-use model and introduces more configuration to the system.

2.3.5 Flint

We move away from MapReduce inspired frameworks and discuss Flint, a Spark execution engine that runs on top of AWS serverless services. This framework allows users to use PySpark, a Python API for Spark jobs, to execute data analytics without the need for a spark cluster. Interestingly, different to every other framework we have discussed, Flint uses distributed data streams as its data exchange mechanism for shuffle intense jobs (Kim et al., 2018). In this section we focus on Flint's use of streams.

Architecture

Flint architecture is shown in figure 8. Flint uses two main components: the scheduler and the executor. The scheduler receives data from Spark's Task scheduler and asynchronously invokes Flint executors as Lambda functions. The scheduler keeps track of the status of the executors and invokes the next set of executors as soon as the last stage is complete. Flint executors can read input data from S3 or SQS depending on the stage they belong to. In figure 8, we have a set of executors in yellow that read from S3 as they are from the first set of executors and subsequently the executors in green read data from SQS (data that was produced from the yellow executors). It is evident that all executors that are not in the last stage write their output to SQS while the last stage writes back to S3.

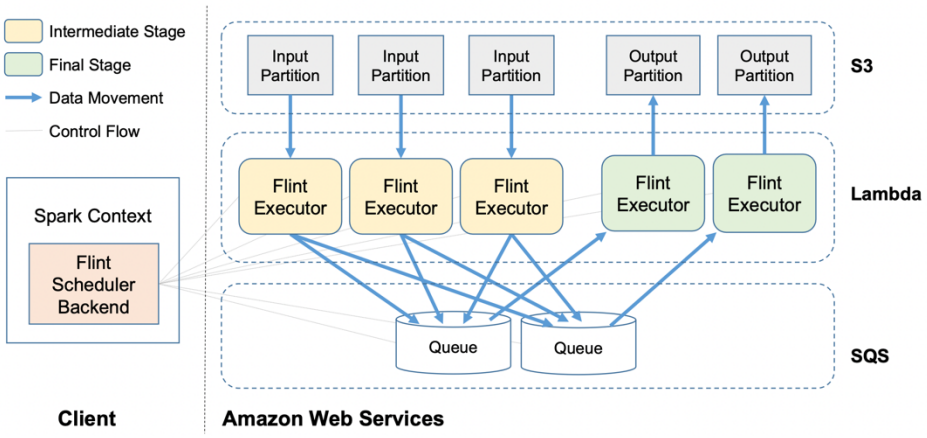


Figure 8: Flint architecture (Kim et al., 2018)

Data shuffling

Tasks that are part of the intermediate stages require their data to be shuffled such that all values from the same partition are grouped together. The partition is achieved by having multiple SQS queues, each receiving a set of partition keys that are generated using a hash partition function. Output from the executors is sent in batches to the SQS queues. If memory is not limited, the batches will be sent at the end of the execution, otherwise executors will spill batches to free the memory. Executors from the next stage will read a queue in a one-to-one relationship and will reduce the results in memory.

Advantages and limitations

- To work around the timeout limit of Lambda functions, Flint executors stop ingesting new data once they reach a specified timeout limit so that they can save their state and continue execution on another invocation of a Lambda function.
- Reducer executors processing the shuffled data need to reduce in memory. Since memory in Lambdas is restricted, Flint makes sure there are enough queues to distribute the data such that no queue exceeds the memory limit, however, if not configured correctly reducers could have a memory overflow failure.
- Given that SQS has an at-least-once delivery mechanism, reduce executors should have implemented a deduplication mechanism. Its absence could lead to incorrect results as duplicated messages could be processed more than once. Implementing this brings an overhead to the memory of the Lambda functions that was not assessed by the framework.

3 Ribble

In this dissertation we introduce Ribble, a serverless data processing framework written in Go that can efficiently process big data stored in the cloud. Ribble focuses on four main aspects: providing an easy-to-use programming model to define jobs, providing a highly scalable and fault-tolerant mechanism transparent to the users, providing a pay-as-you-go billing model, and allowing users to run ad-hoc jobs without having to configure servers.

Easy-to-use programming model

Ribble is a framework tailored for analysts without experience on distributed systems or parallel computing. For this reason, Ribble is designed to offer a simple programming model that allows users to define distributed processing jobs as if they were writing sequential code. Ribble, like MapReduce, is a framework that consists of map and reduce phases. However, Ribble's programming model moves away from the MapReduce paradigm by removing the reduce function definition which allows users to define jobs more naturally. Instead, users write a map function where they are forced to think as if they were writing a sequential program containing all the logic to process a single file. This logic is automatically extracted by Ribble and applied in the reduce phase. Note that to achieve distributed data processing it is not as simple as running the map function in multiple servers where each take a different file, instead an aggregation phase is needed which requires the framework to schedule and synchronize the map and reduce components.

Removing the reduce function definition decreases the flexibility users have to define jobs because users can only choose from a set of aggregators when defining the map function. However, these aggregators enable users to define many analytical jobs and they should be easy to understand as they resemble basic SQL aggregators. Apart from designing the programming model to reduce the complexity to define jobs, Ribble offers a Go Package and a command line interface (CLI) that facilitate its use.

Engine and infrastructure

Moving away from distributed data processing frameworks that run on clusters environments, Ribble explores the feasibility of using serverless functions and streams as its main units of computation and intercommunication. For a Ribble workload, streams offer a natural way for serverless functions to communicate. Because serverless functions have a short lifespan, data does not persist and they cannot be used to communicate with other serverless functions, an event-based mechanism is suggested which allows for serverless components to communicate.

Ribble exploits the offerings from the cloud to create a complete serverless framework which allows users to run ad-hoc jobs without having to configure a single server. Ribble is implemented to run in AWS as it is one of the most mature cloud providers and it offers a complete suite of services, SDKs and tutorials that made developing a cloud framework simpler. Because of the variety of services offered, when designing the framework special attention was needed to choose the best services in terms of cost, speed, and scalability.

Parallelization, scalability, and fault-tolerance

Ribble was designed to allow users to run processing jobs able to efficiently distribute the computation to thousands of servers while transparently handling the parallelization, data distribution, scheduling, and fault-tolerance to the users. While inspiration is taken from MapReduce to achieve such mechanisms, a significant amount of time was invested to achieve this in a serverless architecture.

Pay-as-you-go and zero configuration

Finally, Ribble is built to run in a serverless architecture which allows Ribble jobs to run with minimal setup steps and provide the users with a pay-as-you-go billing model which lowers the cost for running ad-hoc jobs.

3.1 Architecture

Ribble shares components with many of the existing serverless data processing frameworks discussed in the background section. In this section we provide an overview of the components and workflow of a Ribble job.

The driver

The driver is the only component that does not run as a serverless function. Instead, the driver is offered as a command line interface (CLI) and its main functionalities are to build, upload and start the job. Although the driver needs to be executed in the users' local machine, Ribble is still considered a fully serverless framework, as it does not need to be alive for the duration of the job. Instead, after building and uploading the resources, it starts the coordinator, and the driver stops its execution.

The coordinator

The coordinator is a Lambda function that runs for the duration of the job and its main responsibility is to initiate and keep track of the map and reduce phases of the job.

The Mappers

The mappers are the main computational unit of a Ribble job. Mappers are Lambda functions invoked by the coordinator and their responsibility is to process the input data according to the user defined map function and sending this intermediate data to the reducers.

The Reducers

The reducers are Lambda functions responsible for aggregating the streams containing the intermediate data sent by the mapper. They are also responsible for filtering, sorting and writing the final output to S3.

3.2 Workflow

Figure 9 shows the workflow of a Ribble job. Note that this workflow does not consider the steps needed to build a job and creating the resources needed in the user's AWS account. Before the first step described below, Ribble would have created and uploaded the coordinator, mapper, and reducer Lambda functions based on the user defined job. Amongst other resources, the driver would have created an S3 bucket for the job and the SQS queues needed to shuffle data. Many aspects of the workflow are abstracted but a full description is given in the implementation section.

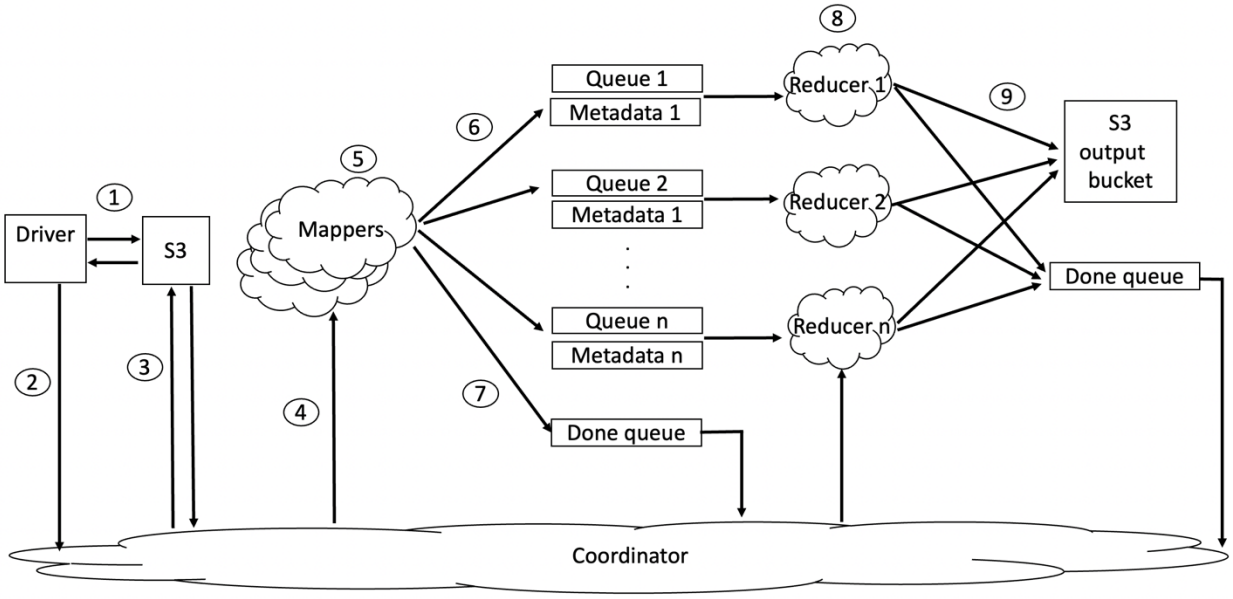


Figure 9: Workflow of a Ribble job

Workflow:

1) The driver queries the S3 input bucket and determines how to split the data so that each mapper gets a similar chunk of data. This data is needed by the coordinator to invoke the mappers, so the driver writes this information back to S3.
2) The driver invokes the coordinator. At this point the coordinator takes over the execution of the job and the driver does not need to be used nor be kept alive as a background process in the user's computer.
3) When the coordinator starts, it reads the input splits from S3 that were generated by the driver.
4) For each split, the coordinator invokes a new mapper.
5) At this point, possibly hundreds of mappers are running in parallel, each processing a different input split. Each mapper downloads the S3 objects it needs to process and runs the user defined function.
6) Running the user defined map function generates a map containing key-value pairs that have already been locally aggregated. To aggregate the values globally, values with the same key need to be aggregated by the same reducer. To achieve this the key-value pairs are sent to different SQS queues according to a partitioning function that guarantees that values with the same keys will end in the same queue. Additionally, each mapper sends a message to each partition indicating how many values it sent to each queue.
7) Once each mapper has finished sending its intermediate data, it sends a message to the done queue indicating it has finished. This queue is read by the coordinator to determine when the map phase has completed. Once it receives a message from every mapper it invoked, it begins the reduce phase by invoking a reducer for each SQS queue.
8) At this point, multiple reducers are running in parallel, each aggregating the values of a single SQS queue. Once a reducer has completed its execution, it will have generated a map of key-

value pairs that are globally aggregated. If included in the job definition, the reducers run a filter and sort function.

9) Once each reducer has finished aggregating, filtering, and sorting the data, they write the output to S3. Because multiple reducers are needed to aggregate the data, the output will consist of multiple objects, each representing a partition aggregated by a reducer. After the output has been written, each reducer will send a done message to the done queue. The coordinator will read this queue to determine when the job was finished.

3.3 Defining a Ribble job

Ribble jobs are defined programmatically with the Go programming language using Ribble's API. This API is offered as a Go package that contains the methods, variables, and data structures that the users need to define a job.

3.3.1 Structure

A job needs to be defined by the user in two Go packages. The first package is used to implement the map, filter, and sort functions. Each of these functions must follow a specific signature and need to be defined as public methods. The second package is used to configure the job and it must be defined as Go executable, that is, it needs to implement a main function and the package of the file must be defined as main.

The following sections describe how a job needs to be defined. As we go along, a word count job is implemented and used as an example. Recall that a word count job is used to count the occurrences of words in the input data, and it is the equivalent of a 'hello world' program for data processing frameworks. The structure below is used for the word count example: wordcount.go implements the map, filter, and sort functions and is included in the wordcount package. job.go defines the configuration of the job and is presented as a main package.

```
.
└── wordcount/
    ├── wordcount.go
    └── job/
        └── job.go
```

3.3.2 Aggregators

Frameworks like MapReduce allow users to define a reduce function that specifies how to process the intermediate data generated by the mappers. While it provides the users with more flexibility, in most cases, the reduce functions only implement simple aggregations and make the framework more difficult to use. To create a simpler and more natural way of defining jobs, Ribble removes the notion of the reduce function and introduces a set of aggregators. Because users can only choose from this set of aggregators, some jobs supported in MapReduce may not be able to be defined in Ribble, however, the aggregators will support a wide range of jobs that require counting, finding minimums and maximums and performing averages.

The aggregators are pre-defined functions that the Ribble API provides, and they specify how key-value pairs in the map and reduce phase need to be aggregated. In the map phase, the aggregators are used to define how the generated values from an input file are aggregated. Like Flink aggregators, in the reduce phase, they define how to transform the incoming data from the streams.

This approach was inspired by Sawzall. Recall from the background section that Sawzall is a programming language that allows users to process logs using MapReduce as its backend. The map function in Sawzall implicitly defines the reduce phase and this is possible because every emitted value from the mappers to the reducers is represented by an aggregator. For example, as shown below, the count variable is initialized as a table sum. When a value of 1 is emitted to count, then it knows it should be aggregated using a sum.

```
count: table sum of int;

emit count <- 1;
```

A similar approach is implemented for Ribble, however the syntax to emit a key-value pair does not exist. Instead, the values are stored in a map which holds key-value pairs, where the type of each value is a Ribble aggregator. In the map phase, every time a new value is added to the map it is automatically aggregated and in the reduce phase, reducers get the key-value pairs from the streams, and they aggregate them according to the aggregator type of each value received.

Instead of implementing the aggregators offered by Sawzall, Ribble is inspired by SQL and implements four of its most popular SQL aggregators. While more aggregators can be implemented in the future, Ribble uses these as they give users enough functionality to run most jobs that generate numeric output.

A short description of the aggregators can be found below alongside examples of how the Ribble Go package can be used to add values to a map. For the examples below, imagine a job which analyses customer purchases in a shop. The examples are set so that the output of reducing such keys is always 6. For example, in the first case, adding the values of the map using the AddSum method indicates that 4 and 2 need to be processed using the Sum aggregator which makes the reduce output to be 6. Note that the variable 'output' in the examples represents the aggregator map which can be initialized using the NewMap function as shown below:

```go
output := aggregators.NewMap()
```

1. Sum

The Sum aggregator is used to add together values that have the same key.

E.g. Get the total number of apples sold:

```go
output.AddSum("Apple", 2)
output.AddSum("Apple", 4)
```

2. Max

The Max aggregator is used to obtain the value with a given key that has the highest value.

E.g. Get the maximum number of oranges bought by a single customer:

```
output.AddMax("Orange", 3)
output.AddMax("Orange", 6)
```

3. Min

The Min aggregator is used to obtain the value with a given key that has the lowest value.

E.g. Get the minimum number of watermelons bought by a single customer:

```
output.AddMin("Watermelon", 14)
output.AddMin("Watermelon", 6)
```

4. Avg

The Avg aggregator is used to calculate the average of the elements with a given key.

E.g. Get the average number of bananas bought by each customer:

```
output.AddAvg("Banana", 8)
output.AddAvg("Banana", 4)
output.AddAvg("Banana", 6)
```

3.3.3 Defining the map function

The map function is used to define the map and reduce phases of a Ribble job. To transparentize how the framework parallelizes each phase and how data is transferred from the mappers to the reducers, Ribble relies on the map function to follow the following signature:

1. Every map function needs to be defined as a public function. This is necessary so that the function can be imported from the map Lambda function. In Go, this is done by setting the name of the function to start with an upper-case letter.

2. The input of the function must be a string and it should represent a file name to process. In this way, the fact that the input data is stored in an S3 bucket is abstracted from the user and users instead need to process the input files as if they were stored in the file system of their computer. With this, users avoid having to use the AWS API to download objects from S3 and instead can use Go's standard libraries. Having this as input, restricts the users to think sequentially as they can only specify how a single file needs to be processed and aggregated.

3. The output of the function needs to be an aggregator map. An aggregator map is a map that stores key-value pairs where the keys are strings, and the values are one of the aggregators defined in the previous section. In MapReduce inspired frameworks, users need to use an emit function to specify what data needs to be sent to the reducers. While it may not be difficult to include as part of the user defined functions, it forces users to understand how the computations are parallelized which can be difficult for inexperienced users. To avoid this, Ribble uses the aggregator map from the output of the function and automatically sends each value to their corresponding streams.

The function below illustrates how a Ribble map function that performs a word count job can be defined.

```
func WordCount(filename string) aggregators.MapAggregator {
```

```
    // open file from the fs
    file, err := os.Open(filename)
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()

    // initialize map
    output := aggregators.NewMap()

    // process lines
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        line := scanner.Text()
        words := strings.Fields(line)
        for _, word := range words {
            // add word to map
            output.AddSum(word, 1)
        }
    }

    return output
}
```

3.3.4 Defining the filter function

While most of the logic for the reduce phase is defined in the map function definition, Ribble supports two additional functions that are applied in the reduce phase. These are the filter and sorting functions.

The filter function is a user defined function used to filter out the values from the aggregator map once all the values have been globally aggregated. For this reason, the function must take as input and output a MapAggregator object.

The function shown below called LessThanFive, is a filter that can be applied to the WordCount job. It shows a filter that removes words that do not appear more than five times across all input files. While this example only shows how to filter by value, filtering can also be done by taking the key or the type of aggregator used. For example, a filter could be written to filter out all words than do not appear at least 5 times and the words that start with the letter ‘a’. As another example, a job that uses multiple aggregators, could use a filter that is only applied when the aggregator is of a given type.

```
func LessThanFive(mapAggregator
aggregators.MapAggregator)aggregators.MapAggregator {
    // delete all items from map that have less then 5 count
    for key, aggregator := range mapAggregator {
        if aggregator.ToNum() < 5 {
            delete(mapAggregator, key)
        }
    }

    return mapAggregator
}
```

3.3.5 Defining the sort function

The sort function is also user defined and it is applied on the globally aggregated values. It is important to note that global sorting can only be achieved if a single reducer is used to aggregate all the values.

To implement the sort function, first the users need to define a list object that implements the 'sort.Interface' from the sort package offered by Go's standard library and then they need to define the function to sort the aggregator map using such interface. The following example shows how the user can write a function that sorts the aggregator map by key and value.

The first step users need to do is to create the list that will implement the interface. To define the list, the Ribble Go package offers the type AggregatorPair which has two fields, a key field of type string and a value field of type float64.

```
type AggregatorPairList []aggregators.AggregatorPair
```

The following step is to implement the 'sort.Interface' using the AggregatorPairList type. Three methods must be implemented: Len, Swap, and Less. Len simply needs to return the number of elements in the collection. Swap and Less define how to swap and sort two elements given their indexes. Because the first two methods are too simple, we focus on the Less method which defines the core logic. The function below illustrates how sorting by key and value can be performed. Given two elements in the list with index i and j, the elements should be sorted by key alphabetically, however, if the elements have the same key, then the elements are sorted according to their value.

```
func (p AggregatorPairList) Less(i, j int) bool {
    if p[i].Key == p[j].Key {
        return p[i].Value < p[j].Value
    }

    return p[i].Key < p[j].Key
}
```

Once the AggregatorPairList implements the sort interface, it can be used to define the sort function. The function below takes as input the aggregator map and returns a list containing the values of the map sorted by using the Sort function provided by Go.

```
func Sort(ma aggregators.MapAggregator) sort.Interface {
    aggergatorList := make(AggregatorPairList, len(ma))
    i := 0
    for k, v := range ma {
        aggergatorList [i] = aggregators.AggregatorPair{Key: k, Value:
v.ToNum()}
        i++
    }

    sort.Sort(aggergatorList)

    return aggergatorList
}
```

3.3.6 Defining the configuration

The configuration can be defined using the Config struct provided by Ribble's Go package. The config below illustrates an example.

```
config := ribble.Config{
   InputBuckets:        []string{"my-input-bucket"},
   Region:              "eu-west-2",
   Local:               false,
   AccountID:           "123456789123",
   Username:            "my-iam-user",
   LogicalSplit:        false,
   RandomizedPartition: true,
   LogLevel:            1,
}
```

The config provides the input data, the AWS account details and configuration details necessary to define how the processing is achieved.

Input: Ribble was implemented to process data stored in S3. Users can define the input data by providing a list of S3 buckets. Currently, Ribble cannot specify or filter objects within a bucket, instead logic can be added in the map functions to filter these.

Account: both the S3 input data and the Ribble resources needed to run the framework must be within the same AWS account. To use this account, its details need to be passed to Ribble. The region field specifies the region where the input is located, to reduce latency and costs the resources are created within this region. The account id and username are the AWS account id to use and the IAM user that will run the job. Note that this information is not sensitive and can be made public if desired. On runtime, Ribble uses the private credentials stored in the user's machine that are generated when configuring the AWS CLI to avoid having to hardcode the credentials. Finally, the Local flag specifies if the job should be processed locally with the environment Ribble provides or within the AWS account. Running jobs locally can be used to test jobs.

Job: finally, the last three elements in the config: LogicalSplit, RandomizedPartition and LogLevel are needed to define how the job should run. How these affect a Ribble job will be explained in a later section, for now it is enough to understand that LogicalSplit defines how the input data is split into chunks so that the computation can be parallelized. RandomizedPartition is used to define the type of partition function to use when sending the intermediate data from the mappers to the reducers. And LogLevel specifies the type of logs that are to be collected, this can be set to info, error, or warning logs.

Putting everything together

Configuring a Ribble job is done via the 'Job' function provided by the Ribble Go package. This function takes as input the configuration of the job and the three functions described above: the mapper, filter, and sort function.

Once the configuration struct has been defined, the user can use the Job function to define the job as shown below:

```
ribble.Job(
        wordcount.WordCount,
        wordcount.LessThanFive,
        wordcount.Sort,
        config,
    )
```

This defines that the mappers should use the WordCount function to process the input data. Once the data is aggregated by the reducers, it defines that the output should be filtered and sorted using the LessThanFive and Sort functions. Finally, the config struct defines the input data, AWS account to use and configuration to be used when running the job.

3.4 The Ribble CLI

Amongst other functionalities, the CLI is used to build, upload, and run a job that has been defined using Go as explained in the previous section. The following section briefly describes the functionality of the CLI commands, and each is accompanied by an example to show how the word count example can be executed.

Set credentials: Because Ribble needs IAM permissions to access S3, SQS, Lambda, IAM, CloudWatch, and ECR to run a job, the set credentials command is used to set the necessary permissions so that the given user within the AWS account can upload and run a job.

```
ribble set-credentials \
    --account-id 123456789123\
    --username my-iam-user
```

Build: The three main units of computation are the coordinator, mapper, and reducer Lambda functions. The build command is used to translate the user defined functions to create Go files and package them as Docker containers that can be used to create the Lambda functions. This command gets as input the job.go file where the word count job was defined and outputs a generated id for the job.

```
ribble build --job ./wordcount/job/job.go
```

Upload: The AWS resources required to run the job are created and uploaded to the AWS account using the upload command. Amongst other task, it includes uploading the Lambda functions generated in the previous step, creating an S3 bucket for the job and SQS queues to hold the intermediate values.

```
ribble upload --job-id <id-of-job>
```

Run: After all the resources have been uploaded the run command can be used to initiate the job.

```
ribble run --job-id <id-of-job>
```

Track: Once a job is in execution, the track command allows users to track it. This command shows the progress of the map and reduce phases and indicates if the job has been completed.

```
ribble track --job-id <id-of-job>
```

4 Implementation

4.1 The Driver

The driver is the only non-serverless component and it is instead offered as a CLI that has five commands: build, upload, run, set credentials, and track. The first three are used to 1) build resources from a job definition 2) upload the resources necessary for the job to AWS and 3) run the jobs. Set credentials enables account administrators to give access to users within their account to use the account resources needed to run Ribble jobs. Finally, track can be used to track the progress of jobs.

The following section is used to discuss the implementation and importance of each of these commands.

4.1.1 Set credentials

In cloud computing, access control is essential to ensure the data, applications, and resources are secure. It usually is not the most secure to give unrestricted access to every user within an account. Users will have distinct roles and hierarchies within an organization which means not everyone needs access to the same resources. For example, if a company stores sensitive data in the cloud, the account administrator should ensure only the necessary and trusted users can access it. Restricting access to resources reduces the surface area for cyber-attacks and makes users within the account less likely to commit errors that can affect the business.

In AWS, Identity and Access Management (IAM) is the service that allows the account owner to control who has access to the resources in their account. IAM allows for granular access control where different permissions can be applied depending on the services, resources and users trying to access it. As the name of the service implies, there are two main components of the service: identifying who is trying to use a particular resource and determining if they have access to them.

To identify who is trying to access a resource, IAM provides two identity objects: users and roles. In one hand, IAM users allow individuals or applications to have access to an AWS account via a set of permanent and unique credentials. In the other hand, IAM roles can be used by any users who need temporary access to resources they would not normally have access to (https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html, consulted on 20/4/2022). In terms of access management, IAM policies are objects used to specify a set of permissions on different resources. It is only when an IAM policy is attached to an IAM identity that AWS can successfully determine if the given entity has access to the resource requested.

By design, Ribble needs access to resources from an AWS account to read the input data and to run processing jobs. An IAM role was created to allow multiple users to run Ribble without sharing credentials or having an account administrator rolling the credentials every time a new user needs to run a job. As well as reducing management efforts, the resources are more secure.

The CLI ‘set-credentials’ command allows account administrators to set up the permissions Ribble needs and allows them to specify which users are allowed to run jobs. This command has four tasks:

1. **Creating a role:** Instead of attaching policies to every user that wants access to the account resources, Ribble creates an IAM role that users can assume.
2. **Creating the policy:** Amongst others, the policy includes permissions to create and invoke Lambda functions, permissions to read and write to SQS and S3, permissions to create ECR repositories and push images, and permissions to write and read logs from CloudWatch.
3. **Attaching the policy:** A role without a policy does not have permission to access anything. In this step, the policy is attached to the Ribble role.
4. **Set 'assume role' permission:** Not every user within the account has permission to assume a role. Each user must have a policy that allows them to assume the Ribble role. To achieve this, the last step of the 'set-credentials' command is to create such a policy and attach it to the specific user defined when running the command.

Note that this command needs to be run by someone who has administrative permissions and before running it, the AWS CLI should have been configured in their local environment. This ensures that the credentials needed by the AWS Go SDK are present and that the user has permission to create the roles and policies.

Once the role with its policy has been created and the user is allowed to assume the role, Ribble uses the AWS Security Token Service (STS) to request temporary credentials that allows the AWS Go SDK to assume the role. This allows Ribble to authenticate with AWS without needing to store any long-term credentials and hence it reduces the risk of having unauthenticated access to the account resources. Note that the driver is the only component that needs to request these temporary credentials as the coordinator, mappers and reducers inherit the credentials from the component that invokes them.

Ribble access resources in AWS using the AWS Go SDK, which contains a set of clients that can communicate with the different services in AWS. An AWS client in Go is configured with a config object which holds a set of credentials. Hence, to configure the clients with the Ribble role, the config object must contain the temporary role credentials provided by the STS service. This mechanism is shown in the code snippet below.

```
// Load the configuration using the aws config file
cfg, err = config.InitCfg(driver.Config.Region)
if err != nil {
    return nil, err
}

// create assume role provider
stsSvc := sts.NewFromConfig(*cfg)
roleArn := fmt.Sprintf("arn:aws:iam::%s:role/ribble", conf.AccountID)
stsCredProvider := stscreds.NewAssumeRoleProvider(stsSvc, roleArn)

// update credentials
cfg.Credentials = stsCredProvider
```

4.1.2 Building a job

The build command of the driver focuses on generating the resources needed to create the coordinator, map, and reduce Lambda functions from the job definition. While the user defined functions provide the framework with the logic needed to process the input data, all the mechanisms needed for the

communication, synchronization, scheduling, and fault-tolerant behavior are missing from the functions. Additionally, the user defined functions do not satisfy the structure needed to create Lambda functions as per the Go specification. The build step is used to translate the user defined functions, to functions that implement the Ribble mechanisms and that satisfy the requirements set by the Lambda programming model.

The four stages for this command are described below:

Stage 1: validate the job

This step checks that the map, filter, and sorting functions used to define a Ribble job follow the signatures required. Given that Go is a typed language, these checks ensure that the functions do not cause any errors when generating the code. If any of the function is not defined correctly, the build step will fail.

Stage 2: generate Lambda code

Once the job definition has been validated, Ribble generates the code needed to create the coordinator, mapper, and reducer Lambda functions. For each function, Ribble needs to generate a Go file that complies with the Go reference to create Lambda functions. Each file must:

1. Define the entry point containing the Lambda Start function from the 'github.com/aws/aws-lambda-go/lambda' package which implements the lambda programming model for go.
2. Implement the handler function responsible of processing events.
3. Be written as a Go executable, meaning that it must have a main function and its package name must also be called main.

To generate the code for each Lambda function, Go's built-in templating package was used. This package is a data-driven library that parses a string template and generates an output file after applying a data struct that contains the template replacement values.

To generate the Go files for the coordinator, map and reduce Lambda functions, three different templates were used, one for each function. Below, we will go through the steps taken to generate the map function:

Step 1: Get the template data

To generate each file from the templates, the driver needs to collect a set of values that will replace the placeholders in the template. This data is acquired from the job definition file that is needed as the input to the command. Recall this Go file contains the map, filter, sort functions and the configuration struct. In the case of the mapper, it uses the job definition to get the package and function name of the user defined map function. From the configuration it gets the partition function to determine how values in the mapper will be shuffled, the Local flag to define how to build the AWS Go clients, and the Log level to configure the logger.

Step 2: generate the Go files

Once the necessary data is collected from the job definition, the driver parses each template with the relevant data to create the Go files. In the case of the mapper, amongst other functionalities, the

Lambda function must define the map function used to process input files. To define this, the following line in the template is used:

```
mapOutput := lambdas.RunMapAggregator (*filename, {{.PackageName}}.{{.Function}})
```

The RunMapAggregator is a wrapper function that takes as input a filename and a map function, runs the map function with the given file and returns an aggregator map. After parsing the template with the data acquired in the first step, it produces the following line where WordCount is the map function.

```
mapOutput := lambdas.RunMapAggregator(*filename, wordcount.WordCount)
```

This templating mechanism is used to generate the code for the three Lamba functions. An example of the auto-generated files can be found in the appendix.

Stage 3: generate Docker images

Once the code of the Lambda functions has been generated, each needs to be packaged and uploaded to AWS. To do so, the Lambda service allows users to upload the code as container images to an AWS Elastic Container Repository (ECR) repository. Note that ECR is simply a service that allows users to store images. This ECR repository is then used by the Lambda service to pull the images when creating containers that execute new event invocations.

The 'build' CLI command is responsible for building the Docker images that package the Lambda functions. Images are templates of read-only layers that set the configuration needed to create containers. Images can be created from scratch, but often are created based on other images. To package Lambda functions, AWS requires the users to build the container from an image that implements the Lambda runtime. The Alpine image was used as the base image in Ribble as it implements the Lambda runtime while offering a minimal Linux container which is only 5 MB in size. To generate the images, the 'build' command goes through two steps: first it generates Dockerfiles for each of the Lambda functions and then uses these Dockerfiles to generate images using the Docker client.

A Dockerfile is a text document that contains the instructions needed to build an image. Dockerfiles are constructed by using a sequence of statements of the form `INSTRUCTION arguments` where `INSTRUCTION` are commands defined and implemented by Docker and arguments are the arguments needed for that specific command. `FROM` is an instruction used to define the base image and takes as argument the name of the image to pull. `ARG` is used to set an environment variable for the container. `WORKDIR` is used to create a directory and jump into it. `RUN` instructs the docker client to run the instruction specified in a shell. `ADD` is used to add resources in the local file system into the container's file system. `COPY` copies resources from a layer below. And finally, `ENTRYPOINT` is used to define the executable to use when running the container.

These Dockerfiles need to be generated every time a new job is built because the Go files used to generate the executables are always different as they define the specific user map, filter, and sort functions. Hence, to generate the Dockerfiles a similar strategy was used as in the previous step, where templates were used.

The Dockerfiles are split into four main components. The first one sets the instructions Docker needs to set the environment where the Go executables are built. The second one adds the Go source files and

dependencies. The third builds and compresses the executable. The fourth defines the entry point of the Docker image.

The size of an image can become a bottleneck for the Lambda functions since larger images will take longer to initialize, reason why a multi-stage build approach, compression tools and a minimal base image were used. Overall, using these techniques reduced the image size from 30 to 8 MB.

Dockerfile part 1: setting the build environment

To build a Go executable, a machine needs to have Go installed and configured. First, the Dockerfile indicates to use the 'golang' image as the base image. This image is the official Go image which contains a Linux distribution with the Go programming language installed. While this image is not minimal, it is only used in the build stage of the Dockerfile which means that its size will not contribute to the final size. Then, it instructs the container to create a directory called 'build' in the root of the filesystem and jumps to it. Finally, using apt-get, the package manager for the distribution, it installs upx. Upx is the executable packer that will be used to compress the Go executable.

```
FROM golang as build
# create work directory
WORKDIR /build
# install tools
RUN apt-get update && apt-get install -y upx
```

Dockerfile part 2: adding the resources

Once the necessary tools are installed, the Dockerfile instructs how to add the Go resources from the local machine that were generated in the previous stage. It starts by adding the dependency files and downloading them. Then it adds files from the Ribble source code needed, these include the pkg and internal directories. Finally, it adds the Go file for the Lambda function generated in the previous step and adds the user's source code where the map, filter, and sort functions are defined. Note that '`./build/lambda_gen/2edae98f-14eb-4386-acdb-7fd9c6bf188b`' is the directory where the generated files are located and `2edae98f-14eb-4386-acdb-7fd9c6bf188b` is the id of the job being built.

```
# add dependancies
ADD go.mod go.sum ./
RUN go mod download
# add Ribble source files
ADD ./pkg ./pkg
ADD ./internal ./internal
# add generated Go files
ADD ./build/lambda_gen/2edae98f-14eb-4386-acdb-7fd9c6bf188b
./build/lambda_gen/2edae98f-14eb-4386-acdb-7fd9c6bf188b
# add user's source code
ADD ./wordcount ./wordcount
```

Dockerfile part 3: building and compressing

At this point, all the resources needed to build the Go executable are within the file system of the image and the correct tooling has been installed. The Dockerfile now defines how to build the Go executable for each of the Lambda functions. The first RUN command is used to build the executable using the 'go build' command. Note it instructs the executable to be built for a Linux machine with the amd64 architecture, which is required by AWS. The second RUN instruction defines how to compress the executable generated using upx.

```
# build lambdas
RUN env GOOS=linux GOARCH=amd64 go build -ldflags "-s -w" -o
/build/lambdas/ ./build/lambda_gen/2edae98f-14eb-4386-acdb-
7fd9c6bf188b/map/WordCount.go
# compress
RUN upx --best --lzma /build/lambdas/WordCount
```

Dockerfile part 4: setting the entry point

Finally, the Dockerfile needs to specify the entry point of the image which is used to define what the container should do when it is invoked. The Lambda service requires the image to have the Go executable that runs the Lambda event listener as the entry point.

To reduce the size of the image, we use a multi-stage build which allows the final image to reduce its size by not using the previous layers. This means the new image will not have the source files, dependencies or even the Go language installed. Instead, a smaller image than 'golang' is used and the executable built in the previous step is copied to the new layer.

```
# Build runtime for map_2edae98f-14eb-4386-acdb-7fd9c6bf188b
FROM alpine as map
COPY --from=build /build/lambdas/WordCount /lambdas/WordCount
ENTRYPOINT [ "/lambdas/WordCount" ]
```

At this point Ribble has generated the Go files for the lambda functions and the Dockerfiles to package each of these. To build the images, Ribble must use the Docker client. While this could have been achieved programmatically by using the Go Docker API, the Docker CLI was used combined with a shell script because the Go Docker API did not support many features and it was not well documented.

4.1.3 Uploading a job

The upload command of the driver focuses on two aspects: determining how to split the input data into similar sized chunks and creating resources on the user's AWS account.

Generating mappings

Ribble uses a single instruction, multiple data approach to process data in parallel and hence exploits data level parallelism. To achieve this, Ribble must find a way to split the input objects into chunks that have a similar size and are small enough to achieve a good parallelization ratio. The size of the mappings is restricted by the memory limit the map Lambda function is configured with. However, this is not a limitation given that having small chunk size allows the framework to do more map operations in parallel. The default configuration for the map Lambda function is 128 MB reason why by default the chunk sizes are 64 MB. This allows the map Lambda functions to store the object in memory while giving enough space for the map functionality to run without reaching a memory limit.

The algorithm used to split the input data can be configured with two modes: using a logical split or not using it. This needs to be specified when defining the configuration of the job by setting the LogicalSplit field to true or false accordingly. These two methods are used in Ribble to give the users more flexibility in terms of the file types they can process. Logical split can be used when the objects do not lose their meaning if they are split. For instance, a text file or a CSV can be split into different parts, each

processed by a different Lambda function and the results would be the same. On the contrary, images would not be a good candidate to use logical splits. Take for example a processing job that uses an image classification algorithm to determine the class of thousands of images. If an image of a dog is split into half and two Lambdas try to classify each part, then it is likely that the job would produce an erroneous output. For these cases, generating the mappings without logical splits can be used however it comes with a limitation, objects cannot be larger than the total size a mapping can hold, that is 64 MB.

For both modes, the algorithm takes as input the name of the input S3 bucket and it outputs a list of mappings. A mapping is used to represent a list of objects that a particular Lambda function will process. Hence, the number of the mappings indicates the maximum number of map functions that can run in parallel. As shown below, the ObjectRange struct is used to represent an element in a mapping where Bucket is the name of the S3 input bucket, Key is the name of a specific object in the bucket and InitialByte and FinalByte represent the range of the object that can fit into to the mapping. Note that if the entire object can fit, then the InitialByte is set to 0 and FinalByte to the total size of the object.

```
type ObjectRange struct {
    Bucket      string
    Key         string
    InitialByte int64
    FinalByte   int64
}
```

The algorithm uses a similar approach to the bin packing problem using a first fit strategy to generate the mappings. While the fit first strategy may not create optimal mappings, in general they will have similar sizes, and this is enough to achieve good parallelization. First fit was used because of its simplicity.

The splitting algorithm starts by getting a list of objects in the S3 bucket. It then loops through the objects to decide if they should be included in the current mapping or start a new one. If the entire object fits in the current mapping, then it is added. If the current object does not fit in the current mapping and LogicalSplit is being used, then it figures out where to split the object such that a part of it goes into the current mapping and the rest goes into the next mapping. In the contrary, if LogicalSplit is not being used and the current object does not fit in the current mapping then, the object is added to the next mapping.

To figure out where to split a file, the algorithm uses the end of line character ‘/n’ reason why only file types like CSV and text files are supported to use LogicalSplit. To find how to split a file, the algorithm first gets the available space in the current mapping which indicates the maximum size of the split to generate however this split is not taken directly because it does not guarantee that the split does not cut a line in two. To find exactly where to split the file, it downloads the last 50 Bytes of the object hoping to find an end of line character. In most cases, it will be able to find one but in case it does not, it retries by downloading the last 100 Bytes and so on, doubling the size each time until it finds an end of line character. Once found, the byte at which the end of line character was found is used in the FinalByte field of the ObjectRange struct which will indicate the Lambda mapper to download the object from the initial byte to the last end of line character.

Once a list of mappings has been generated, it is written to an object in S3 called ‘mappings’. This object is downloaded by the coordinator Lambda function and it uses it to invoke the mappers. While the

mappings could be sent to the coordinator as a parameter when invoking it, the size of the invocation payload is restricted to 256 KB which could become an issue as the input grows.

Creating resources

Ribble uses the AWS Go SDK to create the following resources into the user's AWS account:

S3 resources: an S3 bucket for the job is created. This bucket is mainly used to write the output of the job however it is also used to write checkpoints generated when reducers process the messages from the mappers and is used to write the mappings generated in the previous step.

CloudWatch resources: a log stream is created for the job. This log stream is used by the coordinator to log the progress of the job.

ECR resources: the images for the coordinator, mappers and reducers generated in the build command are uploaded to an ECR repository. To achieve this the driver first creates an ECR repository for each of the functions.

SQS resources: for each reducer it creates two queues, a message queue used by the mappers to send their key-value pairs to the reducers and a metadata queue used by mappers to indicate how many messages they sent to each queue. If the job runs using randomized partitions, then a final aggregation step is needed, and an extra message and metadata queues are added.

Additionally, it creates two queues to track the progress of the map and reduce phases. The mappers and reducers send messages to these queues whenever they are finished so that the coordinator can keep track of the progress.

Finally, two dead-letter queues are created. Dead-letter queues are SQS queues used to send messages that cannot be processed and are helpful to the users to debug issues that occur when running a job. The first dead-letter queue is used by the Lambda service, and it is configured such that the Lambda invocation events that failed at least three times are sent as messages to this queue. This means that the Lambda dead-letter queue will contain information of failed mappers or reducers. The second dead-letter queue is used to send the events that could not be processed by the reducers.

Lambda resources: Once the images have been uploaded to ECR, Ribble creates a Lambda function for each. The default configuration for the Lambda functions is set to 900 seconds timeout, 128 MB of memory size for the mappers and 512 MB for the reducers, 512 MB of temporary storage and each Lambda is set to restart on failure up to three times.

4.1.4 Running the job

Starting the job is the last step needed to run job. The CLI command 'run' allows users to start the job and it is composed of a single instruction: starting the coordinator. To make Ribble completely serverless it was important to create a driver able to initialize the job asynchronously.

4.1.5 Tracking a job

While not necessary to run jobs successfully, Ribble offers the CLI 'track' command to track the progress of a job. This command uses the CloudWatch API to retrieve the logs generated by the coordinator and outputs them to the standard output. Although not necessary this process can run for the duration of

the job given that the logs are being generated in real time but note that the user can terminate this command and restart it at any point. By default, this command polls the logs every 5 seconds but can be configured to poll as often as needed.

4.2 The Coordinator

A Ribble job is composed of two parts, the map and reduce phases and it is the responsibility of a coordinator to schedule and track each of these.

4.2.1 The map phase

The coordinator starts by downloading from the job's S3 bucket the mappings that were generated when building and uploading the job. The coordinator loops through the elements of this list and invokes a new Lambda map function for each item, allowing the framework to process the input data in parallel.

Originally, the mappers were invoked by the driver using the run command of the CLI. However, this significantly restricted the number of map Lambda functions that were running in parallel and hence slowed down the overall running time. Figure 10 shows the result of a benchmark which compares the maximum number of map Lambda functions running in parallel when invoked by the driver and the coordinator. This was tested with four different workloads to assess how it behaves as it scales. In a perfect scenario, the driver and coordinator should invoke as many functions in parallel as there are mappings however, this was not the case. As the workloads scale, neither the coordinator nor driver can keep up with the number of requests however, the coordinator achieves a significant improvement. When running the 100 GB workload, the Ribble job takes 138 seconds when invoking the mappers from the driver as compared to 37 seconds when doing it from the coordinator. This improvement is attributed to the invocation requests limits set by AWS which allows 10 concurrent invocations per second for resources that are outside AWS such as the Ribble driver. On the contrary AWS allows unlimited invocations from an AWS resource such as the coordinator which runs as a Lambda function.

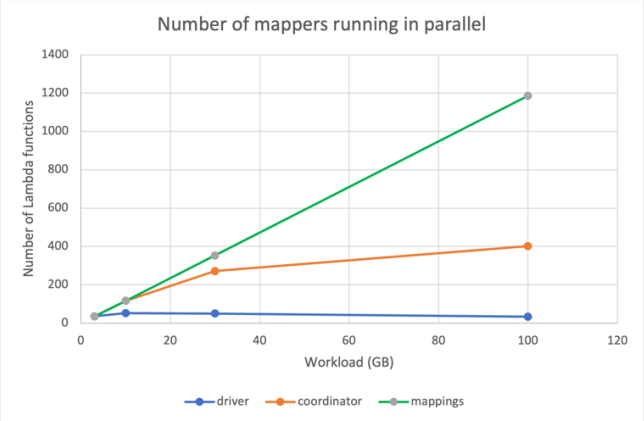


Figure 10: invocations from the driver vs coordinator

Once the mappers have been invoked the coordinator needs to track the progress of the mappers to determine when they have completed running. This is needed so that the coordinator can start the reduce phase only after all the intermediate values have been sent to the SQS queues.

When the coordinator is invoked, it gets the number of mappers it needs to invoke, and this is used to determine when the mappers are finished. Every time a mapper finishes successfully, it sends a message to the map done SQS queue. This queue will contain a message from every mapper once the map phase

is done. The coordinator starts by polling this queue, if no messages are found then it sleeps for a second and retries again. As soon as it receives a message it starts polling messages without going to sleep until it receives as many messages as there are mappings. Given that every SQS request is billed, the sleeping mechanism is used so that the coordinator does not send and receive thousands of empty API calls while the mappers are still running which is especially important where the map function takes a long time to complete.

4.2.2 The reduce phase

The reducers are invoked like the mappers with the difference that instead of invoking a Lambda function per mapping, the coordinator invokes as many reducers as there are queues, such that each reducer aggregates the values from a single queue in parallel. To determine when the reducers have finished, the coordinator follows the same approach as for the mappers but using the reducers done queue.

The job is completed if the hash partitioning was used as it ensures the values have been globally aggregates in each of the reducers. On the contrary, if the job uses random partitioning, then it means that the messages aggregated by the previous reduce phase need to be aggregated again to guarantee that all values with the same keys have been aggregated together. If this is the case, the coordinator begins an extra reduce step. To do this it invokes a single reduce Lambda function and waits until it finds an output file in the job S3 bucket. The random partition behavior will be explained in following sections, for now it is important to understand that the coordinator invokes an extra reduce step.

4.2.3 Logs

Apart from coordinating the map and reduce phases of a Ribble job, the coordinator is responsible of giving users a way to track the progress of a job. Figure 11 shows an example of the logs generated by the coordinator. To log events, the coordinator uses the CloudWatch Go SDK and the log stream that was created by the driver. As shown below, the coordinator logs every time it starts a new phase and logs the number of mappers and reducers that are yet to finish. Finally, it indicates when the job has completed.

```
INFO[0000] Coordinator starting...                       Timestamp="54274-02-25 18:16:51 +0100 CET"
INFO[0000] Waiting for 1 mappers...                      Timestamp="54274-02-25 18:16:51 +0100 CET"
INFO[0000] Mappers execution completed...                Timestamp="54274-02-25 19:44:27 +0100 CET"
INFO[0000] Waiting for 1 reducers...                     Timestamp="54274-02-25 19:44:27 +0100 CET"
INFO[0000] Reducers execution completed...               Timestamp="54274-02-25 21:09:32 +0100 CET"
INFO[0000] Waiting for final reducer...                  Timestamp="54274-02-25 21:09:32 +0100 CET"
INFO[0000] Final reducer execution completed...          Timestamp="54274-02-25 22:33:56 +0100 CET"
INFO[0000] Job completed successfully, output is available at the S3 bucket c9d612e6-c09e-4e65-bffa-ab5590e1a1f6...
```

Figure 11: example of logs generated by the coordinator

4.2.4 Fault tolerance

Because the coordinator runs in a single server, the probability it has of crashing is less than the mappers and reducers that can be running in thousands of servers at the same time. For this reason, some of the frameworks avoid implementing a crash recovery system and simply fail the job if the coordinator fails.

Ribble was designed to have a fault-tolerant coordinator for two reasons. First, the coordinator in Ribble is more prone to crashes as it runs as a Lambda function in a commodity server. This combined with the

lack of control users have to choose specific servers or the type of applications that it shares a server with makes the coordinator to be unreliable. Second, Ribble was designed from its beginning to be fault-tolerant and for this reason adding a crash recovery mechanism for the coordinator was simple and it did not increase the cost as only four additional API requests to the S3 service were required.

In case of a failure the coordinator is configured to retry its execution three times. Given that the function would most likely start in a different server, three attempts are enough to avoid the coordinator to crash because of faulty hardware. While a simple mechanism that re-executes the whole coordinator is possible, the coordinator was implemented to skip the phases it had already done. This is important to save resources, because a failed coordinator does not mean that the invoked lambda functions stopped. For instance, if the coordinator has to re-execute the mapper Lambda functions, then twice the number of Lambda functions and messages to the SQS would be generated.

To achieve a fault-tolerant coordinator, Ribble relies on S3 to persist data and re-execution of the SQS messages. The following section describes how the coordinator implements a crash recovery mechanism.

When the coordinator starts the mappers, it writes a blank S3 object called ‘mappers-invoked’ to the job’s bucket. This object is used to understand if the mappers have been invoked by a previously crashed coordinator. If this object is found, then the coordinator can skip invoking the mappers.

As explained before, to determine if the mappers have completed their execution, the coordinator reads messages from the ‘mappers done’ SQS queue and uses a counter to count the number of mappers that have completed. In case of a crash, the coordinator does not rely on saving this counter as a checkpoint. Instead, if it fails, the coordinator can re-play the messages from the queue. This is possible because the SQS queue was configured to use a visibility timeout of 30 seconds. This means that after 30 seconds, the messages once read by the crashed coordinator can be processed again. Because each mapper produces a single message to this queue, re-playing the messages does not cause a large overhead.

The reduce phase is implemented in a similar way. The coordinator checks for an S3 object called ‘reducers-invoked’ to determine if the reducers have been invoked by a previously crashed coordinator. And like, the map phase, to check if the reducers have been completed is simply re-reads the messages from the reducers done queue.

4.3 The Mappers

The mapper is the serverless component responsible for processing the input data with the user defined map function. Recall that the map phase is parallelized by splitting the data and distributing the splits to multiple instances running the map function. Each mapper gets a list of S3 objects to process and for each object they follow the steps described below:

Step 1: downloading the object

The object is downloaded to a temporary directory '/tmp' in the file system of the Lambda function. This is done by using the S3 download manager provided by the AWS Go SDK which allows the mapper to download an S3 object using a specified range. This is necessary because a split assigned to a mapper may contain only a part of an object if the original object was too large for a single mapper.

Step 2: executing the map function

The user defined function is executed using the object downloaded in the previous step. Recall that Ribble requires the map function to take as input a string which represents the name of a file in the local file system. When running the map function, this parameter is set to the name of the object currently being processed.

By design, Ribble has local aggregation ingrained in the mapper execution. Local aggregation allows for a reduce-style mechanism that runs in the map phase and it is used to reduce the number of values emitted to the reducers which makes shuffling data more efficient. In Hadoop MapReduce, local aggregation can be achieved by using combiners. Combiners are a class provided by Hadoop that passes the values emitted by the map function and aggregates the values by using the reduce class in the same machine where the map was executed. Apart from combiners, local aggregation can also be achieved in Hadoop by explicitly modifying the map logic to account for local aggregation.

In contrast, Ribble users do not need to consider how to aggregate the values locally. Instead, this happens automatically when adding or updating a value in the MapAggregator which holds the values to emit. Take as example the following snippet where the first line initializes a MapAggregator, and the following two lines add the "Hello" key with two different values using the Sum aggregator:

```
output := aggregators.NewMap()
output.AddSum("hello", 1)
output.AddSum("hello", 5)
```

When the second value is added with a key that already exists, the "hello" entry of the MapAggregator is updated by adding 1 plus 5. Similarly, if the AddMax method was used, the entry would have been updated to 5. While this example is good to understand how local aggregation is implemented it does not show why this would be useful. Take for instance a Ribble job that counts the number of times the word 'hello' appears in one thousand text files. For simplicity, assume that each mapper is responsible of processing a single file. If the word 'hello' is found a thousand times in each file then, without local aggregation the mapper would emit one million messages as opposed to one thousand using local aggregation.

Step 3: shuffling data

After executing the map function, the mapper holds a MapAggregator object that contains the key-value pairs that have been already locally aggregated. The mapper now needs to emit these values to the reducers. This step is referred to as data shuffling.

In general, serverless frameworks struggle to achieve an efficient shuffling mechanism and this is because client-server communication protocols cannot be achieved using serverless functions. Frameworks that use a cluster of computers to run MapReduce workloads rely on the Remote Procedure Call (RPC) protocol to communicate between the mappers and reducers so that the intermediate data stored in the file system of the mappers can be used by the reducers.

This type of client-server communication protocol cannot be implemented when working with serverless functions because of two reasons. First, the data within a Lambda function cannot persist after it has finished its execution, even if it is saved to the function's file system. This means that to allow the reducers to communicate with the mappers using a client-server mechanism the mapper needs to be kept alive. Because Lambda functions are limited by an execution timeout, keeping them alive means that less capacity is given to the mappers to execute the data processing operations. Another implication of keeping the mapper functions alive is that the cost will increase significantly as Lambda functions are charged by the second. Second, even if keeping Lambdas alive was cost efficient, they are not designed to communicate with each other. To implement an RPC protocol between mappers and reducers, each mapper and reducer needs to know each other addresses and this is not possible with the Lambda service. For this reasons, serverless frameworks require an external service to store the intermediate data.

To store the intermediate data, S3 is the first service that comes to mind, as it is commonly used by serverless frameworks running MapReduce workloads as it is a service with pay-as-you go billing method, it can scale quickly on demand, and it is the service that resembles a file system the most which is used with traditional distributed data processing frameworks. Frameworks relying on a distributed file system can take advantage of the data locality to place mappers or reducers that are closest to the input or intermediate data. However, using S3 this is not possible as AWS does not allow users to choose the servers. While S3 data could be stored in the same server running a Lambda function, this is completely up to AWS.

Instead of using an object storage service like S3, Ribble explores using a distributed event-based system inspired by Flint which uses data streams as its main data exchange mechanism. Because Ribble processes static data, it deals with bounded data streams. Using a distributed event-based system, allows the Lambda components that were not designed to interoperate to work together (https://learn.lboro.ac.uk/pluginfile.php/1682291/mod_resource/content/4/lecture07-pubsubsystems.pdf, consulted on 20/4/2022). An event-based system allows for asynchronous communication which allows the mappers to send their intermediate values without waiting for a response from the reducers hence allowing both components to have independent lifetimes. It also allows for a many-to-many communication mechanism which allows multiple mappers to send their data to multiple reducers at the same time. Finally, the mappers and reducers do not need to know the location of each other to communicate. These characteristics make the mappers and reducers in Ribble to be time and space uncoupled which allows the framework to scale.

SQS as the event driven mechanism

While there are other services that could have been used to achieve an event driven mechanism, SQS was selected because it is offered as a serverless service which is highly scalable and offers a pay-as-you go billing model. AWS kinesis was a service we considered when designing the shuffling mechanism, and while it offers better throughput if configured correctly, each stream is charged by the second and is not able to scale on demand.

Recall that in the driver section, it was discussed that the upload command is responsible of creating the SQS queues needed to send the intermediate data from the mappers to the reducers. In this section we discuss how an event-based mechanism is achieved using these SQS queues.

An event-base system is composed of producers, consumers, and an event broker. In Ribble, the producers are the mappers, the consumers are the reducers, and the event broker is the SQS service itself, who is responsible of scaling the queues to meet the demand and handle the API requests to send or receive messages. A channel-based approach was used where mappers produce values to multiple queues and the reducers subscribe to a single queue, thus making each queue a partition that hold all the intermediate values that belong to a set of keys.

Ribble's ability to scales comes from the fact that the mappers and reducers can scale and run in parallel. Increasing the number of mappers allows Ribble to process more input data in parallel and increasing the number of reducers allows it to aggregate more intermediate values in parallel. It is important to notice that increasing the number of mappers and reducers does not necessarily decrease the processing time of a job, instead a balance needs to be achieved.

Typically, a Ribble job will require less reducers than there are mappers because aggregating data is usually faster that processing the input data. The number of reducers in Ribble is hugely influenced on the size of the intermediate values generated by the mappers because a reducer needs to store in memory all the values it is responsible of aggregating. This means that the number of reducers needs to be chosen so that the data in any given queue is not higher than 512 MB, the amount of memory configured for the reduce Lambda functions. Because mappers are configured with 128 MB, the default number of reducers set for a Ribble job is a quarter of the number of mappers. This configuration will allow a Ribble job to work when the size of the intermediate values generated by the mappers equals the size of the input data.

Reducers in Hadoop MapReduce do not have this memory limitation because of a sort phase introduced before the reducers begin aggregating data. This means that each file is already sorted by the time a reducer reads from its input partition and this allows each reducer to apply a merge sort algorithm that globally sorts all the files in its partition. With these values sorted, a reducer can iterate through the intermediate values and start a reduce task as soon as it sees a value with a different key that the one it is currently aggregating.

This sort mechanism is powerful, however, it is not possible using standard SQS queues because they do not guarantee the messages to be processed in the same order as they were sent, which means that even if the data was sorted in the mappers once sent via SQS the data could have been reordered. It is important to note that this sort mechanism could have been achieved using FIFO SQS queues as they guarantee first-in-first-out ordering. However, FIFO SQS queues are limited to 300 API calls per second compared to almost unlimited throughput for standard SQS queues.

Partition functions

While setting the correct number of reducers is critical so that the reducers can process all data in memory, it is fundamental that good load balancing is achieved in the queues. To achieve this, Ribble introduces two partition functions: hashed and random partitioning. Partitioning functions are used to determine the partition any given key-value pair needs to be sent to.

Hash partitioning is a common algorithm used by several of the frameworks reviewed in the background section and it is used to choose a particular partition based on the key of each key-value pair. In Ribble, this function is implemented using the crypto/md5 Go package to encrypt the keys using the md5 algorithm. Once encrypted, they are encoded as hexadecimal strings and its integer representation is used with a modulo operation to determine which partition the key should go to. This allows the keys to behave like a random distribution with the benefit that a particular key will always end up in the same partition.

Figure 12 illustrates how the mappers send intermediate values to the streams using the hash partitioner. Notice that all the values with keys A, B or C are sent to the first stream and values with keys D and E are sent to the last stream.

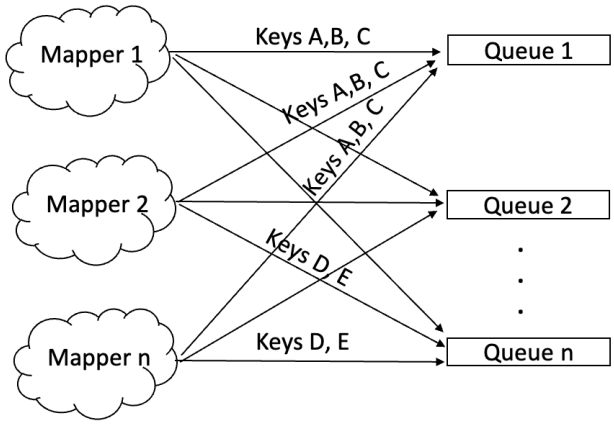


Figure 12: shuffle mechanism using the hash partitioner

This hash partition function creates good load balancing if the keys in the MapAggregator map have a good distribution. While in many data processing applications this is possible, it is not always the case. For instance, take a job that wants to aggregate a single feature, if the hash partition function was used then all the values generated from the mappers would be processed by a single reducer. To address this problem Ribble offers the random partitioner. The random partitioner is used to assign a random partition to each key. To implement this function, Go's math/rand package is used to create a number generator. To create a different sequence of random numbers for each mapper, a seed is generated by encoding the map id with the md5 algorithm and because every map id is randomly generated it creates random seeds. The random partition function allows for good load balancing even when the keys are not well distributed however keys with the same value may end up in different partitions. To address this a final reducer responsible of aggregating all the keys is needed, this will be discussed in the reducer section.

Emitting values to partitions

To send the messages from the mappers to the queues, the mapper loops through the MapAggregator map, applies the partition function with the current key to determine the partition it should be sent to and uses the Go SQS client to send the message to the SQS queue. To send a key-value pair of the MapAggregator, the mapper needs to transform the pair into a JSON object that SQS can handle. The ReduceMessage struct shown below is used to marshal the key-value pair information into JSON where the Type field is used so that the reducer receiving the messages knows how to aggregate them, the EmptyVal field is used to indicate if a message is empty, and the Count field is used when sending messages of type Avg.

```
type ReduceMessage struct {
    Key      string
    Value    float64
    Count    int
    Type     int64
    EmptyVal bool
}
```

Standard queues promise at-least-once delivery rather than exactly-once. This means the messages could be sent multiple times to the queues and the mapper needs to take this into consideration to make the application idempotent. Idempotency refers to the ability of the application to handle repeated messages or events such that the application behaves as if they were processed only once. This is achieved using a deduplication algorithm in the reducer, which will be fully explained in the reducer section. For now, it is important to know that the mapper adds sequence numbers to the metadata of each message. Specifically, it adds the id of the mapper sending the message, the id of the batch and the id of the message within the batch. These values alongside the ReduceMessage JSON are used to represent the SQS messages.

To improve the throughput of sending SQS messages, the mapper was implemented to send messages using the batch operation provided by the AWS Go SDK. This operation allows to send 10 messages with a same API call, hence improving the throughput up to 10 times. To achieve this, the key-value pairs generated by the mappers are grouped such that keys that need to be sent to the same partition, which is indicated by the partition function, are grouped into the same batch. Once a batch reaches 10 messages, the batch is sent to the given queue. This allows the mapper to have a memory flush mechanism and hence not all values need to be stored for the duration of the map function.

Step 4: send metadata

A reducer finishes its execution once it has processed all the messages in its partition. To determine when a reducer has processed all the intermediate values is not as simple as stopping as soon as it stops receiving new messages from its queue. This is because SQS standard queues offer eventual consistency which means that some messages may take longer to appear in the queues and if those messages appear after a reducer has finished processing all the available messages, the reducer will skip values which will make the job to output an incorrect result. Another reason why this is not possible is because messages may be hidden according to the visibility timeout they were configured with. If a message was not processed correctly for any reason, the message will go back to the queue and will be available after

a certain time however, reducers may have finished processing the available messages by the time the failed message becomes available again.

For these reasons, each reducer needs to know exactly how many messages they are supposed to process. This information is sent from the mappers to the reducers using metadata queues. Each partition has a corresponding metadata queue where each holds the number of messages sent to that queue by each mapper. How the reducer handles these messages will be described in the reducer section.

Step 5: send finished event

Finally, the last step of the mapper is sending a done message to the mappers done queue. This message contains the map id of the current mapper, and it is used by the coordinator to determine if all the mappers have completed.

Fault tolerance

Fault tolerance for the mappers is based on re-execution. Like the other Lambda functions in the framework, the map function is configured to run up to two times more in case of a runtime error. Because the mapper function emits values to the partition streams, the framework must ensure that it is able to handle repeated data. Recall that because of the at-least-once behavior of standard SQS queues, a mechanism to handle duplicate events is already in place and it can be used in the same manner to detect messages that were sent twice because of mapper re-executions.

However, to ensure the dedupe mechanism in the reducers work as expected, the partition functions must ensure that their behavior does not change if the mapper re-executes. The hash partitioner has no problem because it always generates the same partition for each key. The random partitioner is different because the algorithm depends on a random number generator. However, recall that this number generator is based on a seed generated from the mapper id. Because the algorithm to create the seed will always create the same seed if the same id is used and a re-executed mapper retains the same id, the values will always be sent to the same random partitions.

Apart from this re-execution mechanism a dead-letter queue is configured with the mapper Lambda function. The dead-letter queue is used to send the invocations that made the mapper crash after two re-executions and can be used to debug the map function. It can also be used by the users to determine if a job which failed to process a mapping is valid or not. This may be the case if the users require an estimate and only 64 MB were not processed out of 1 TB of the input data.

4.4 The Reducers

The reducer is the last of the components in Ribble and it is the main unit of computation for the reduce phase. The reducers responsibility is to aggregate the intermediate values generated by the mapper and applying the filter and sort user defined functions. There is a one-to-one relationship between partitions and reducers which means that a reducer is responsible of aggregating a single partition and this is specified by the coordinator when it invokes each reducer. Below we describe the step a reducer takes to aggregate the data:

Step 1: get number of batches to process

To determine when a reducer has completed processing the messages in its partition, each reducer must know how many messages the mappers sent. Unluckily, the only API call to achieve this is a call that produces an approximate value because standard SQS queues are eventually consistent.

To overcome this, the reducer loops through the messages in the metadata queue for its partition and calculates a sum of all the values sent. Note that because these messages are sent via SQS, messages can be duplicated so a deduplication mechanism is needed to make the reducer idempotent. Because each mapper must send exactly one message, this algorithm is simple to implement. Each message in the queue contains the id of the mapper who sent the message and the total number of batches it sent, and the reducer keeps a map data structure that indicates the mappers it has already received the data from.

The reducer stops reading from the queue when it has read as many different messages as there are mappers in the current job. This value is passed to the reducer from the coordinator when the reducer is invoked.

Step 2: receive messages

Once a reducer knows how many messages it needs to aggregate, it enters a loop until all values have been aggregated. For each iteration of the loop, the SQS client in the reducer requests the SQS service for new messages. The reducer receives up to ten messages per call and for each message it must determine if it has been processed before or not. Recall that each message sent by the mappers contains the mapper id, batch id and message id as metadata. These values are used by the deduplication algorithm described below.

Deduplication

The deduplication mechanism is built upon a deduplication map which is used to determine if a message has been already processed or not. The deduplication map is a map with three levels, or in other words a map of maps of maps. While this sounds complicated, it was designed like this to achieve memory and access efficiency. The first level of the map represents the mappers, the second map represent the batches sent by each mapper, and the third represent the messages within each batch.

This map is presented in Go as a type with the following signature:

```
type DedupeMap map[string]map[int]*DedupeProcessedMessages
```

Where the DedupeProcessedMessages is used to represent the messages within a batch using the struct shown below:

```
type DedupeProcessedMessages struct {
    ProcessedCount int
    Processed      map[int]bool
}
```

Where ProcessedCount field is used to store the number of messages that have been processed in the batch and the Processed field is a map used to indicate if a message within the batch has been processed already.

For example,

```
dedupeMap["eeebb1b5-a775-49f5-94bd-5b83828cdf9b"][1].Processed[2]
```

Can be used to determine if the second message of the first batch that was sent by the mapper with id eeebb1b5-a775-49f5-94bd-5b83828cdf9b has been processed or not.

Similarly,

```
dedupeMap["eeebb1b5-a775-49f5-94bd-5b83828cdf9b"][1].ProcessedCount
```

Can be used to determine the number of messages that have been processed for the first batch that was sent by the mapper with that id.

On startup, the reducer creates an empty DedupeMap. As it receives new messages, it gets the metadata of the messages and checks weather the current values exist in the dedupe map and skip the messages if it is present in the map. This allows the reducer to identify duplicated messages in three operations.

To optimize the memory usage, the deduplication algorithm uses the ProcessedCount field of each batch to determine if the information of a batch can be deleted from the map. Recall that the mappers send the messages in batches of exactly 10 messages. This means that once a reducer has processed 10 different messages from a batch it can delete the information of the individual messages as any new messages that have this batch id can be safely ignored.

Step 3: process messages

Recall that each message in a stream will be presented as a JSON object containing the key, value, aggregator type and an 'empty' Boolean field. Each message will be unmarshalled into the ReduceMessage struct, which if you recall, is the same struct used by the mapper when sending the messages.

On startup, the reducer function initializes an aggregator map which will be used to hold the aggregated data as new intermediate values are processed. A reducer acts in three different ways depending on the message received:

1) If the message is empty, it is simply skipped.
2) If the key of the message is not in the aggregator map, the reducer creates a new key-value pair for this key where the value's type is set to the type specified in the message. Adding the correct

type for the value is necessary so that new values with that same key that are added to the map are aggregated correctly.

3) If the key of the message is in the aggregator map, the reducer adds the new key-value pair using the Reduce method of the specific aggregator. Each aggregator implements the Reduce method, an example for the Min aggregator is shown below. It simply takes the message and check if the value in the aggregator map should updated, which is only the case if the new value is smaller.

```
func (m *Min) Reduce(message *ReduceMessage) error {
    if m.ToNum() > message.Value {
        // update new min
        m.Min = message.Value
    }

    return nil
}
```

After each message is processed, the reducer updates the message counter which used to determine if more messages need to be processed. If this counter matches the value calculated in step 1 then it breaks the receive messages loop.

Step 5: filter

After all the messages have been processed, the aggregator map will hold all the intermediate values grouped by key and aggregated accordingly. In the filter step, the user defined filter function is applied to the MapAggregator map. This filter function takes as input the MapAggregator map and outputs the same object without the entries that were filtered out.

Step 6: sort

Whether or not the filter function was used in the previous step the reducer encounters an aggregator map and sorts it according to the user defined function. It is important to notice that this function sorts the data locally.

Step 7: write output

After the intermediate data has been aggregated, filtered, and sorted the results need to be written to S3. Each reducer writes its output to a single object in the job's S3 bucket. Like MapReduce, Ribble will generate the output in multiple partitions, depending on the number of reducers used to process the intermediate data. While the output of a job is split, Ribble guarantees that the values are globally aggregated, and it is left to the users to define how they consume the output of the job.

Step 8: send finished event

Finally, once a reducer has finished processing a stream, it needs to indicate to the coordinator that it has completed successfully. Like the mappers, this is done using the reducer done SQS queue where each reducer sends a message with their id included.

Implementing the random reducer

In the map section, it was discussed the random partitioner function that can be used to emit intermediate data from the mappers to the reducer where the keys of the intermediate data do not follow a good distribution.

Notice the main difference between the hash partitioner, which is used in the workflow explained above, and the random partitioner is that the reducers processing the streams do not get all the values associated with a particular key. This means that the values would not be globally aggregated if the same workflow was followed for jobs using the hash and random partitioners. To address this, the workflow for jobs using the random partitioner are divided into two reduce phases: a random reducer and a final reducer. Figure 13 illustrates the data shuffling mechanism needed to aggregate the values globally. Notice how the mappers do not sent keys with the same value to the same partition. For instance, queue 1 and 2 both aggregate values that belong to the key 'B'.

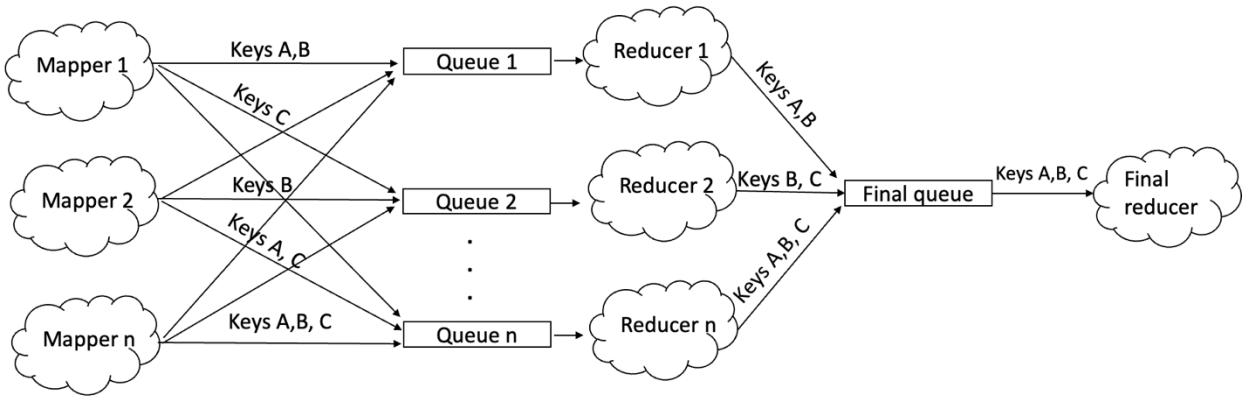


Figure 13: shuffling data using the random partitioner

The first reduce stage shown above, is used to aggregate the random intermediate values emitted to it and has a similar workflow as the one explained above with the main difference being that at the end, the aggregator map is not written to S3. Instead, the reducers behave like mappers in the sense that they send the aggregated values to the final reducer via SQS. Because only one reducer is needed to achieve global aggregation, the mechanism is easier than the one explained in the mapper section as it does not need to determine a partition to send the data. In terms of the metadata, each random reducer sends a message to the metadata queue for the final reducer which includes the reducer id and the number of messages it sent. Finally, the random reducers emit a done message to the reducers done queue that enables the coordinator to determine when to start the final reducer. The final reducer is used to aggregate the values sent by the random reducers and ensure that all the values with the same key are aggregated together.

Running a Ribble job with the random partition function gives the users more flexibility while it improves the scalability of jobs where the keys are not well distributed. Without the random reducer, the job would process all the intermediate data with a single instance however, the random reducer allows for an extra aggregation phase which can reduce the number of messages the final reducer needs to process.

Fault tolerance

The reducer is the component where fault tolerance is the most difficult to achieve. Like other Lambda functions, the reducer relies on re-execution to overcome crashes, however, a simple re-execution mechanism is not enough.

Ribble is limited by the messages in-flight limit, set by SQS, which restrict the number of messages that have been read but not yet deleted to 120,000 messages. Having a limit on the number of in-flight messages means that the reducer needs to continuously delete messages after processing them from the queue. If this happens, using a simple re-execution mechanism would not produce the correct result because it would not be able to re-play the SQS messages that had already been deleted.

To overcome this issue a checkpointing mechanism is proposed. Checkpointing is a mechanism where the program state is saved in regular intervals and is used so that after a crash failure, the program can resume its execution where it stopped (https://learn.lboro.ac.uk/pluginfile.php/1690389/mod_resource/content/24/Lecture08_FaultTolerance.pdf, consulted on 20/4/2022). To avoid creating unnecessary checkpoints, the reducer is configured to save a checkpoint after it reaches 100,000 in flight messages. With this, the reducer avoids having to delete each message and create a checkpoint after each time a new message is processed.

Saving the checkpoint after 100,000 messages instead of 120,000 messages allows the reducer to start a background thread responsible of creating the checkpoint and deleting messages from the queue while it continues to process new messages. This allows the reducer to continue processing messages from the queue instead of having to stop and wait which improves the performance of the reducer. Giving 20,000 messages as buffer should be enough so that the reducer saves the checkpoint before reaching an in-flight message limit error. However, if a reducer is still in the process of saving the checkpoint and it is close to reaching the in-flight message limit, it stops until the background thread has completed.

Because of the deduplication mechanism the reducer also needs to save the deduplication map. For efficiency the reducer does not save the entire aggregator and deduplication map at every checkpoint, instead only the values that were updated since last checkpoint are saved. This reduces the amount of data that needs to be written, however, a reducer reading the checkpoint data after a crash will need to merge all the checkpoint files rather than just reading the last one. To achieve this, the aggregator and deduplication maps consists of two maps, one for the values that have been saved in a previous checkpoint and the other holding the values yet to save.

To implement the mechanism previously mentioned, Ribble uses Go routines and different synchronization mechanism. Go routines are lightweight threads managed by the Go run time that allows functions to execute concurrently. Go routines share the same memory space and hence, a synchronization mechanism should be considered to avoid creating inconsistencies. A Go routine is defined using the ‘go’ keyword. For example, a function called ‘foo()’ can be executed as a go routine by executing it as ‘go foo()’. In terms of synchronization, the reducer makes use of Go’s built-in ‘sync’ package which provides synchronization primitives like locks and WaitGroups. Locks are used to safely access data that many go routines may be using at the same time while WaitGroups are used to wait for multiple go routines to finish.

Once the reducer reaches 100,000 in flight messages, a series of go routines are launched to create a checkpoint. Specifically, three set of go routines are needed.

1) Two go routines save the unsaved values from the aggregator and deduplication maps to the job's S3 bucket under the 'checkpoints' key. This key is used when a reducer starts to determine if it is recovering from a crash, or it is its first invocation.
2) A go routine is used to delete the in-flight messages from the SQS queue.
3) The last set of go routines merges the unsaved values of aggregator and deduplication maps.

To synchronize the go routines and being able to determine if the checkpoint has been saved, the reducer uses a WaitGroup. Before starting each go routine, the reducer increments the WaitGroup counter to indicate the number of go routines in progress. Each function is written so that at the end of their execution they decrement the WaitGroup counter. Using this counter, the reducer can check the WaitGroup to determine if saving the checkpoint has been completed before continuing processing new messages.

4.5 Logs and errors

In cloud computing systems, logs allow users to get insight into the behavior of a system in real time. Logs are fundamental to identify and fix errors and collect data that can help to determine the performance of the system and how to optimize it. Having a log strategy is essential when working with serverless architectures because data does not persist once they finish their execution either because of a failure or successful completion. Because Ribble has multiple serverless components that can fail at any time, each component was implemented so that in case of an error the users could go over the logs of a components and understand what went wrong.

To store the logs from Ribble CloudWatch was used. CloudWatch is a monitoring and observability service offered by AWS that allows users to explore, visualize and analyze logs of services running in AWS or on-premises. CloudWatch was chosen because it integrates with AWS Lamba and offers an API that can be used with the AWS Go SDK.

To log errors to CloudWatch, first the errors need to be identified. To achieve this, Go allows functions to return errors. This is a common pattern used in Go programs and it enables the component running a function to determine if an error occurred. For example, a function defined with the following signature 'func foo() error' specifies the return type to be an error and can be called as 'err := foo()' where err is a variable that will take the error value or nil if no error occurred. The component calling this function can then check if calling 'foo()' generated an error with an if statement that compares the variable err with nil.

To explain how Ribble sends errors to CloudWatch, we will go through the execution of the SendFinishedEvent function which is used by the mapper Lambada function to send a message to the done queue to tell the coordinator when it has finished. The SendFinishedEvent shown below gets the done queue URL and then uses the SendMessage function provided by the AWS Go SDK to send a REST call to the SQS service. The SendMessage function returns the response of the REST call and an error if one occurred. In this case, the SendFinishedEvent uses the error returned from the AWS call as its return parameter.

```
// SendFinishedEvent sends an event to the mappers-done queue to indicate
// that the current mappers has finished processing
func (m *Mapper) SendFinishedEvent(ctx context.Context) error {
    // get queue url
    queueName := fmt.Sprintf("%s-%s", m.JobID.String(), "mappers-done")
    queueURL := GetQueueURL(queueName, m.Region, m.AccountID, m.local)

    // send message
    _, err := m.QueuesAPI.SendMessage(ctx, &sqs.SendMessageInput{
        MessageBody: aws.String(m.MapID.String()),
        QueueUrl:    &queueURL,
    })
    if err != nil {
        return err
    }

    return nil
}
```

The mapper uses this function definition to send the done message to SQS as shown below:

```
    // send event to queue indicating this mapper has completed
    if err := m.SendFinishedEvent(ctx); err != nil {
        mapperLogger.WithError(err).Error("Error sending done event to stream")
        return err
    }
```

If the function returns an error, the mapper logs this error and provides a user-friendly message. The message is important in this case because the error generated by the SQS service does not add any Ribble business logic which makes it difficult to identify where and why this error occurred. For instance, if the user finds a 404-error message from SQS because the queue does not exist, users may not have enough context to determine which queue caused the error and fixing or reporting the error may become difficult. Additionally, the mapperLogger is a structured logger that includes the job id and map id so that users can identify exactly the component that failed.

The following JSON object represents an example of an error logged when calling the SendFinishedEvent that is found in CloudWatch:

```
{
    "timestamp": 1651491262786,
    "message": "time=\"2022-05-02T11:34:22Z\"
        level=error
        msg=\"Error sending done event to stream\"
        Job ID=c4c812bb-0b9e-462b-848e-69eb52fed0f9
        Map ID=4c9b54f9-af77-4d88-b31d-abfe9b083aa8
        error=\"com.amazonaws.services.sqs.model.AmazonSQSException: null (Service:
AmazonSQS; Status Code: 404; Error Code: 404 ; Request ID: null)\"",
    "ingestionTime": 1651491262808
}
```

With this, the user can know the exact time the error happened, the mapper id that generated the error and the job this mapper belongs to. Additionally, note the user friendly and error messages provided that can be useful for debugging.

The mapLogger is able to upload the logs to CloudWatch automatically because Lambda is integrated with CloudWatch such that each instance is given a log stream where all the data produced to the standard output is written to. These logs can be viewed in real time but they also persist in CloudWatch depending on how the user configures CloudWatch. While Ribble does not offer a utility to view these logs, the users can directly use the CloudWatch console within AWS or use the AWS CLI to get the logs.

5 Using Ribble locally

Ribble relies on AWS services which means that users require to have an AWS account configured and be willing to pay from the first job they run. This has two consequences; potential new users may be reluctant to adopt the framework and current users lack a platform to test their new jobs. For this reason, Ribble offers a local environment that allows new adopters to test the framework without having to configure their AWS accounts and while not designed to process huge amounts of data, it enables current users to test the correctness of their jobs.

To simulate the AWS services locally, Localstack was used. Localstack is a service offered as a Docker image that provides endpoints to most of the popular AWS services, including S3, SQS and Lambda. While most services can be used with the free version of Localstack it excludes the use of ECR and the ability to create Lambda functions from Docker images. Instead, the free version allows Lambda functions to be created from zip files. To make local testing free of charge, the code base was updated to allow the creation of Lambda images from zip files when deploying the jobs to Localstack.

1. Configuring the environment

Docker Compose is a tool developed by Docker and it allows users to spin up Docker resources using the docker-compose CLI and a YAML configuration file. To facilitate setting up Localstack, the Ribble repository provides a Docker Compose YAML file that contains the configuration needed by Localstack, including the image and container names, the ports, volumes and environment values to use and the Docker network to create. Users can use that file out of the box and use the command 'docker-compose up' to spin up the Localstack container.

Once Localstack is running, users need to create an S3 bucket in Localstack and upload their input data to it. This can be done with awslocal, a wrapper of the AWS CLI provided by Localstack that points to the Localstack instance or by using the AWS CLI and setting the endpoint URL flag to Localstack.

2. Running the job

Once Localstack is running and the sample data has been uploaded to S3, the Ribble CLI can be used to deploy and run jobs locally. While the CLI is used in the same manner as if the jobs were to be run in AWS, the configuration of the jobs must account for this by setting the Local flag to true. This flag is used when configuring the AWS Go SDK by pointing the endpoint resolver to Localstack.

The code snippet below, shows an example configuration. Note that Localstack does not check for IAM permissions, which means that the username and account IDs are not relevant and can be ignored. However, the 'set-credentials' CLI command can still be used locally to create the Ribble IAM role and its IAM policy. While not functionally useful, it allows new users to understand how easy it is to configure their AWS account to run Ribble jobs.

```
// define config for a local job
config := ribble.Config{
    InputBuckets:        []string{"my-input-bucket"},
    Region:              "eu-west-2",
    Local:               true,
    LogLevel:            1,
    AccountID:           "000000000000",
```

```
        Username:            "my-iam-username",
        LogicalSplit:        true,
        RandomizedPartition: true,
    }
```

After a job finishes the output can be retrieved from the Localstack S3 service in the same way it would be retrieved from AWS.

6 Testing

For users of big data processing frameworks, it is indispensable to be assured the framework has been thoroughly tested. In one hand, encountering an error after hours of processing would mean a waste of time and money resources. In the other hand, a framework that shows no errors but generates the incorrect output can be even more damaging. For these reasons, testing was considered as fundamental in the development of Ribble.

Unit testing

Unit testing was used to perform validation and defect testing of individual program units and functions. These tests were added during the development of the framework and were instrumental to ensure new functionality and changes in the code base were working as desired and that they were not affecting the existing behavior.

Unit tests were developed using Go's standard testing package alongside Testify and Mockery, tool kits written in Go that provide assertions and mocking functionalities. Assertions were used to test the function responses while mocks allowed us to isolate the functions' behavior from external dependencies by replacing them with objects that simulate their behavior. Given the extensive use of the AWS Go SDK, testing using mocks was essential.

In Go, mocks are created based on interfaces. Like Java, Go interfaces are collections of method signatures that allow programmers to define the methods a type must implement. To satisfy this requirement, the codebase was designed using the factory pattern, a design approach where interfaces are provided for creating objects. In general, happy path tests were used to test the correctness of functions while unhappy paths tested the behavior when functions were set to fail.

Figure 14 in the appendix shows how the function GetNumberOfBatchesToProcess was tested using an SQS mock. GetNumberOfBatchesToProcess is a function used by the reducers that gets the number of messages a particular reducer needs to process based on the metadata received via SQS. In the test, the call to ReceiveMessage is simulated three times by defining an expected input and output. The first two calls simulate a duplicated message from the first mapper which should be handled by the dedupe mechanism and the second message simulates a single message by the second mapper. Finally, the number of batches is compared to the expected number, and it asserts that no error occurred.

Because of time constraints in the development of this project, the test coverage is not too comprehensive and there was no static testing introduced, however, both are tasks that would increase the quality of the framework in the future.

Integration testing and continuous integration

Integration testing was used to provide a more comprehensive coverage of the framework. Given the number of different components that communicate with each other, it was essential to perform integration testing to assess their compatibility. Big bang is an integration testing approach where the complete software system is used to perform the tests. This method was chosen as it allowed us to perform a complete functional test of the system in the most time efficient manner.

Continuous integration (CI) is a software development practice where changes to the codebase are tested regularly, usually after committing to the repository. This allows for automated testing and enables developers to notice errors as early as possible. To enable continuous integration, we used Jenkins, an automation engine that allows developers to configure CI/CD pipelines via scripts. In the absence of a VM to run the pipeline, Jenkins was configured to work locally. The pipeline was integrated with Git such that new commits to the main branch of the repository triggered a new Jenkins build. This ensured that the main branch of the repository was healthy after every change.

Figure 15 shows the 5 stages of the CI pipeline:

1. Checkout code: In the first step of the pipeline, Jenkins creates a working directory and checks out the master branch from the Git repository.
2. Setup: The second step is used to set up the environment. To avoid using AWS to run every test, as discussed in the sections above, a Localstack container is created automatically using Docker Compose. The step concludes after a health check to the Localstack container passes indicating that it is ready to accept requests.
3. Testing: The third stage is to run the tests. The tests consist of building and running Ribble jobs with different configurations from start to finish. Every time a job finalizes, the output of the job stored in S3 is downloaded and compared to the expected output. While errors in the tests are difficult to debug, the benefit is that if passing, we can be confident that all components in the framework are working.
4. Cleanup: In the last step the Localstack container and Docker resources are removed, and the workspace created by Jenkins is deleted.

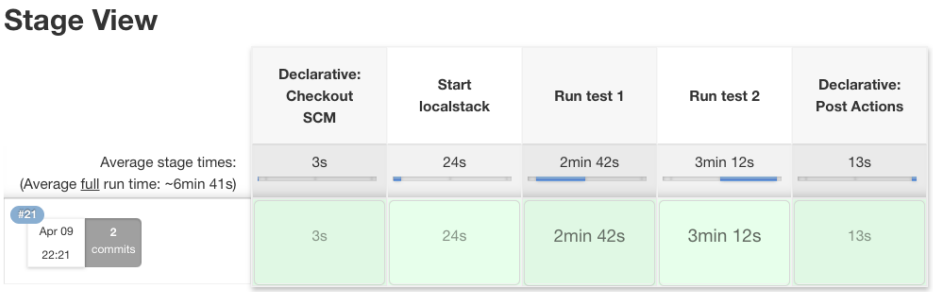


Figure 15: stage view of the CI pipeline

7 Evaluation

The TPC Benchmark H (TPC-H) is a decision support benchmark that consists of ad-hoc business focused queries. The data used by TPC-H simulates relevant industry data and the queries give answers to critical business questions. TCP-H was chosen to evaluate Ribble's performance as it is a recognized benchmark used by worldwide companies like Hewlett-Packard Enterprise and Dell. Additionally, TCP-H was selected because multiple serverless data processing frameworks have been evaluated using this benchmark which allows for a good comparison with the existing work.

Queries

To evaluate the framework, Query 1 (Q1) and Query 6 (Q6) from the TCP-H benchmark were used. These queries were selected given that they are the queries evaluated by other serverless frameworks (Werner et al., 2020) and this allows Ribble to be compared to the existing work.

Q1, referred to as the pricing summary report query, reads data from a single table, 'lineitem', and reports a summary of the amount of business that was billed, shipped, and returned as of a given date. For reference, the query is implemented below as a SQL query. This query allows us to test the performance of a Ribble job that uses multiple aggregators and the filter and sorting functionalities. Note that the parameter 'DELTA' is set so that at least 95% of the rows are used.

```
SELECT
    l_returnflag,
    l_linestatus,
    SUM(l_quantity) as sum_qty,
    SUM(l_extendedprice) as sum_base_price,
    SUM(l_extendedprice*(1-l_discount)) as sum_disc_price,
    SUM(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
    AVG(l_quantity) as avg_qty,
    AVG(l_extendedprice) as avg_price,
    AVG(l_discount) as avg_disc,
    COUNT(*) as count_order
FROM
    lineitem
WHERE
    l_shipdate <= date '1998-12-01' - interval '[DELTA]' day (3)
GROUP BY
    l_returnflag,
    l_linestatus
ORDER BY
    l_returnflag,
    l_linestatus;
```

Q6, referred to as the forecasting revenue change query, uses the 'lineitem' table to get the revenue that could have been generated if some of the discounts were not given to some companies. For reference, the query is implemented below as a SQL query. While this query is simpler than Q1, it allows us to test Ribble using the random partitioner.

```
SELECT
    SUM(l_extendedprice*l_discount) as revenue
FROM
    lineitem
```

```
WHERE
    l_shipdate >= date '[DATE]'
AND
    l_shipdate < date '[DATE]' + interval '1' year
AND
    l_discount BETWEEN [DISCOUNT] - 0.01
AND
    [DISCOUNT] + 0.01
AND
    l_quantity < [QUANTITY];
```

Metrics

To assess the performance of a framework TCP-H requires two main metrics: query-per-hour (QphH) and price-per-hour (CpH). The former measures the framework's throughput capability by indicating how many queries the framework can handle in one hour while the latter reports the costs of running such queries. Because serverless frameworks use a pay-as-you-go billing strategy, using per-hour metrics allows us to compare Ribble with other frameworks that run in managed clusters.

To conduct the evaluation, compute and I/O data were collected via AWS CLoudWatch Logs. Specifically, the coordinator, mappers and reducers were modified to log the number of S3 and SQS operations each performed. Additionally, the number of Lambda functions and their corresponding billed duration and used memory were reported in the logs. This data was used to calculate the running time and costs of Q1 and Q6.

Experiment design

To fairly compare the performance results of Ribble with the frameworks evaluated by Werner (Werner et al., 2020) the experiment was replicated as close as possible. AWS Lambda was configured to run functions with 1535 MB of memory and 512 MB of temporary storage. Note that the AWS EMR cluster used in Werner's experiments was configured to use three AWS EC2 m5.xlarge- instances, two for the workers and one for the master.

The DBGEN tool, offered by TPC, was used to generate the datasets. In total, four different datasets were used, each of a different size. The first dataset of size 3 GB was used to compare the results while datasets of size 10, 30 and 100 GB were used to evaluate the scalability of Ribble. Each dataset was stored in S3 in the region 'eu-west-2', the same region Ribble was configured to run in.

To ensure consistent results, each experiment was conducted three times and the averages were used. Additionally, each experiment was performed at least 15 minutes after each other to ensure the Lambda functions were tested from a cold start.

For reproducibility, the experiment queries can be found in the project repository.

Comparison

For their evaluation (Werner et al., 2020), they use AWS EMR and three different serverless frameworks: Quoble, Corral and PyWren. The results taken from their experiments can be seen in figure 16 alongside Ribble results.

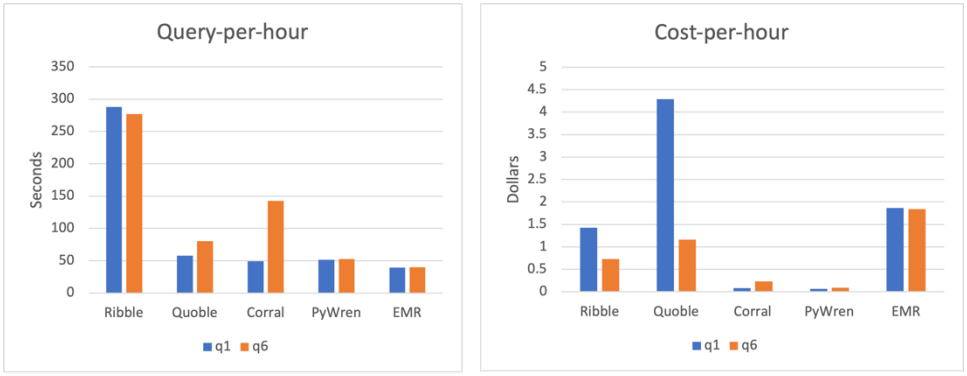


Figure 16: Ribble Comparison of QphH and CpH with data taken from (Werner et al., 2020)

Overall, Ribble performs significantly better in terms of queries-per-hour than the other frameworks, which means that Ribble is the fastest for both Q1 and Q6. Determining the exact reason is difficult, especially when the implementations, programming languages and packages used to develop each framework vary so much. It is also important to note that the results from Werner's results (Werner et al., 2020) were recorded in December of 2020 and Ribble's performance could be attributed to new and faster hardware running the Lambda service or improvements to the Lambda runtimes.

In terms of data shuffling, Quoble, Corral and PyWren use S3 as their data exchange mechanism. In contrast, Ribble uses SQS queues apart from reading the input data and writing the final results. For both Q1 and Q6, Ribble performs the third most I/O operations, which indicates that transferring data through SQS for such workloads leads to a throughput improvement. However, this is only possible because Ribble uses local aggregation. Without it, an estimate of 18 million messages would have been needed to be processed through SQS. Taking into consideration that SQS has 30 thousand messages per second throughput, it would have taken Ribble at least 17 seconds to send the data from the mappers to the queues which is long time considering the job took 12.5 secods.

Interestingly, figure 16 shows that Ribble is the only framework in which Q1 outperforms Q6 in terms of running time performance. This is attributed to the fact that Q1 uses the hash partitioner while Q6 uses the random partitioner where an extra reduction step is needed to aggregate the values from each reducer. The random partitioner was used because Q6 uses a single group value, the revenue, which means that the revenue calculated by each mapper would have been sent to a single reducer, making this reducer aggregate all the values. In contrast by using the random partitioner, the revenue calculated by each mapper was sent to a random queue and hence to different reducers. Although this does not achieve a perfect distribution, it allows the mappers to send more messages without reaching an SQS throughput error and allows the reducers to aggregate the data in parallel. However, the final reduction step increased the running time by 5 seconds.

While Ribble achieved the best results in terms of running time, it was outperformed by Corral and PyWren in terms of costs. The costs of Ribble mostly come from the number of Lambda functions and the memory each function uses. Using the coordinator as a Lamda function allows Ribble to be a fully serverless framework, however, because it needs to be active for the entirety of the job, it creates an extra cost to account for. Because most Lambda functions used by Q1 and Q6 are map functions, reducing the number of mappers would decrease the cost of Ribble. This can be achieved by increasing the chunk size each mapper takes from the input data. An experiment was performed using 256 MB

instead of 64 MB as the chunk size and it led to a reduction of mappers from 35 to 9, reducing the cost by around 150 micro dollars. However, it also increased the runtime from 12.5 to 21 seconds as the data processing parallelism was reduced. This result indicates that users can configure Ribble to run jobs faster or cheaper.

Another factor to account for is the memory used by the functions. While the functions were configured to use 1536 MB to perform the tests in a fair way, none of the Lambda functions in Ribble used above 128 MB. Using this configuration would decrease the cost of Ribble from 498 to 188 micro dollars in the case of Q1 and from 264 to 41 micro dollars in the case of Q6.

Finally, it is worth noting that Ribble outperforms the cluster running AWS EMR in both in throughput and cost. This may not be the case always, especially for shuffle intensive jobs, however it is a good indication that Ribble can outperform frameworks running in cluster environments.

Scalability

Tables 3 and 4 show the performance of running queries Q1 and Q6 with different workloads. To assess the scalability of the framework, both queries were tested with 3, 10, 30 and 100 GB.

Q1: Pricing Summary Report									
Workload	Mappers	Reducers	S3 read ops	S3 write ops	Sqs ops	RT	Cost	QphH	CpH
3GB	35	4	49	5	3894	12.5	498	288	1.43
10GB	116	4	130	5	11184	15	1284	240	2.48
30GB	353	4	367	5	32514	14	4369	257	11.23
100GB	1186	16	2407	17	375020	37	25430	98	24.74

Table 3: Q1 Results

Q6: Forecasting Revenue Change Query									
Workload	Mappers	Reducers	S3 read ops	S3 write ops	Sqs ops	RT	Cost	QphH	CpH
3GB	35	4	49	1	452	13	264	277	0.73
10GB	116	4	130	1	1424	14	650	257	1.6
30GB	353	4	367	1	4268	15	1972	240	4.73
100GB	1186	16	2407	1	42824	26	7031	138	9.73

Table 4: Q6 Results

In terms of speed, Ribble can scale almost perfectly for both queries up to the 30 GB workload. Notice that between the first three jobs there is only one second difference in terms of the running time. However, when running the 100 GB workload both queries get a significant running time increase of 23 and 11 seconds respectively.

This increase is caused by the map phase of the jobs. Tables 3 and 4 show the number of Lambda functions needed for each query. Not surprisingly, the number of functions increases as the workload increases since more mappers are needed to process the input data. Ideally, regardless of the workload, these mappers should run in parallel and should finish at the same time. However, Ribble is limited to burst concurrency limits which refers to the maximum number of instances that can be running at the same time. AWS region ‘eu-west-2’ supports 500 burst concurrency and it increases by another 500

each minute. However, as none of the queries takes longer than a minute to complete the map phase is restricted to 500 instances in parallel.

Figure 17 shows the map phase concurrency each workload achieved in each of the experiments for Q1. The blue line shows the ideal concurrency needed to run the map phase in parallel and the orange line shows the actual concurrency levels for each of the workloads. It is clear to notice that up to 30 GB, the ideal and actual concurrency levels almost match, which causes the running times to be almost identical as almost all the input data processing is happening in parallel. However, as the workload increases to 100 GB, the ideal and actual concurrency levels differ significantly. While 1186 instances were needed to achieve perfect parallelism only 402 were created. This is attributed to the invocation throughput achieved by the coordinator. In the background section, a recursive invocation strategy was mentioned where the first set of mappers are invoked by the driver and each of the mappers recursively invokes more mappers. This strategy could help more instances to be running at the same time and it would be worthwhile investigating as part of any future work.

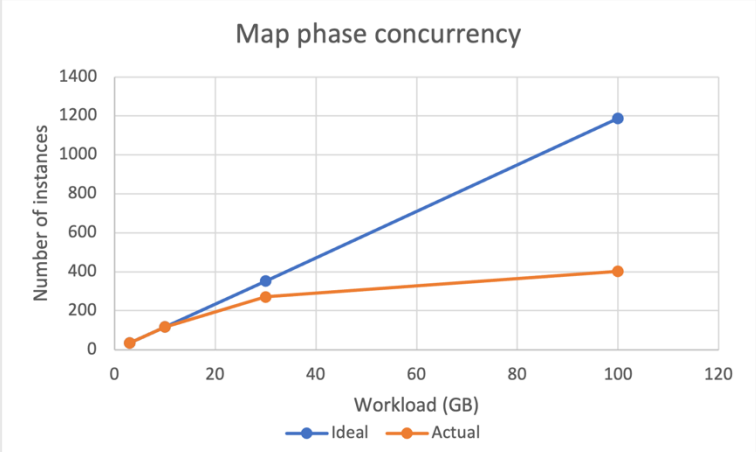


Figure 17: map phase concurrency for Q1

Finally, we notice that Q6 outperforms Q1 as the workload increases. In the comparison section it was discussed that Ribble was the only framework where Q1 outperformed Q6 and it was caused by the extra reduction step needed to aggregate the random partitions. However, this last reduction step becomes unnoticeable compared to the overall running time as the workloads increase.

Figure 18 shows the costs for Q1 and Q6 as the workload increases. As opposed to the running time, the costs appear to increase almost linearly. This is not surprising given that the resources required to run the jobs, including the Lambda functions and SQS messages linearly increase. This linear behavior allows users to predict the costs of running the queries when increasing the input data.

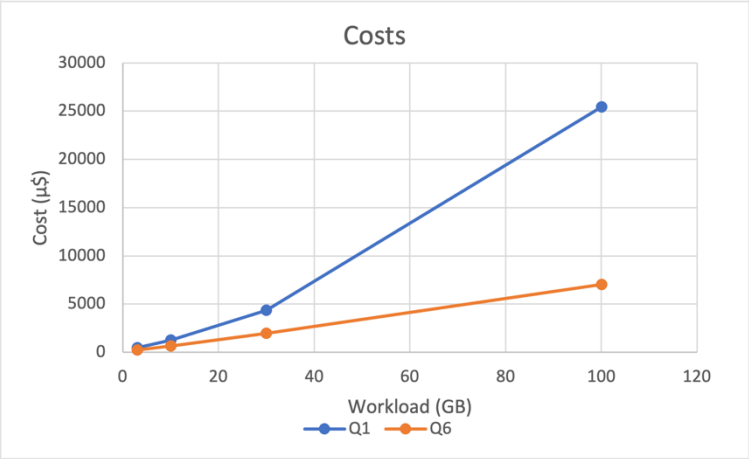


Figure 18 Q1 and Q6 costs as the workload increases

Limitations

So far, it has been assessed that Ribble appears to be the fastest of the four frameworks when using Q1 and Q6 while still offering accessible costs. However, this may not be the case with different queries so it is important to understand the type of queries Ribble can handle efficiently. To assess different scenarios, the focus of this assessment is to understand how the cost is affected by shuffle intensive jobs, jobs in which the mappers require to send a lot of data to the reducers.

In Ribble, shuffle intensive jobs occur when a large number of groups are used for the aggregation. Ribble's ability to perform local aggregation before sending data through SQS, allows each mapper to send as many SQS messages as there are groups, regardless of the amount of data there is to process. For example, when using Q1 to process 3 GB, each mapper gets around 500 thousand rows to process which can be locally aggregated to eight values, hence, reducing the number of messages sent through SQS drastically. In this example, the messages sent by all mappers incur a cost of 0.0015 dollars however without local aggregation it would have incurred at least 1.4 dollars. As the workload increases to 100 GB with 600 million rows to process overall, the price of sending that many messages without local aggregation would go up to $48 as compared to $0.15.

While this was possible for Q1 and Q6, it is not always the case. Take for instance the following aggregation query taken from the Amp Lab Big Data Benchmark designed by Berkley University (https://amplab.cs.berkeley.edu/benchmark/, consulted on 10/4/2022):

```
SELECT SUBSTR(sourceIP, 1, 8), SUM(adRevenue) FROM uservisits GROUP BY SUBSTR(sourceIP, 1, 8)
```

It reads data from a 'uservisits' table and sums the revenue each IP group generates. This query generates 2,067,313 different groups in 25.4 GB of data. If Ribble needs around 400 mappers and each mapper gets around 250 thousand different IP groups, then sending the data from the mappers to the reducers via SQS would generate a cost of $46 making it the most expensive framework assessed in this report.

8 Conclusion

In this project we introduced Ribble, an easy to use yet expressive and scalable data processing framework. In this section we evaluate the degree to which Ribble achieves its requirements, focusing on how it achieves a framework that: 1) is easy-to-use, especially for data analysts without much programming experience 2) can express a wide range of data processing jobs 3) is able to meet the demands of big data and is cost efficient.

Easy-to-use

Ribble's programming model allows users without much programming or parallel computing experience to define job because it forces users to think sequentially, and its syntax hides the fact that the framework runs as a distributed system. With such a programming model, users do not need to be aware of the distributed mechanisms and workflow of the job which makes Ribble easier as compared to other serverless frameworks that use the MapReduce programming paradigm.

Because Ribble runs on top of a serverless architecture, it allows users to run processing jobs without setting up a single cluster. Although it still requires users to configure their AWS account, the CLI provided by Ribble can achieve this with a single command and needs to be only once. In general, the CLI allows users to build, upload, run and track jobs with simple commands.

Finally, Ribble's ease of use is also attributed to the local environment it provides so that users that do not have an AWS account or do not want to use their resources can still use Ribble to test their jobs.

Expressiveness

Ribble's ease of use restricts users from defining the same range of jobs that can run in MapReduce. However, it can be expressive enough for a wide range of data processing jobs. For instance, a Ribble job can achieve the same as a SQL command that uses: SELECT, FROM, WHERE, GROUP BY and ORDER BY using the Sum, Count, Min, Max and Avg aggregators. However, Ribble jobs will be able to process the data using more complex logic and can use all the libraries provided by Go to achieve this.

Efficiency

Ribble was implemented as a fully serverless architecture which allows users to pay exactly for what they use which makes Ribble cost efficient specially for users that require a data processing framework to run ad-hoc queries.

In terms of its ability to scale, the evaluation section revealed that it is efficient in terms of throughput and cost for some queries however it is important to notice that Ribble may not be the best option to run jobs that require many features to be extracted from the input data. This suggests that using data streams as the shuffle mechanism can outperform frameworks that use object storage, however, unless a cheaper stream service can be found, an alternative framework should be used for shuffle intensive jobs.

Future work

Because of the time constraints and the nature of this project, there are features missing from Ribble that would improve the framework. We mention a few in this section:

1. **Support for different data sources:** currently Ribble can only work with input data stored in S3. Providing other data sources would be beneficial to the users that store their data in other services. Because Ribble runs on top of a stream engine, it would be interesting to assess Ribble's capability to process data coming from streams.
2. **Security:** for jobs that need to process sensitive data it would be important that this data remains encrypted when it is transferred from the mappers to the reducers.
3. **Logging utility:** recall that Ribble logs to CloudWatch. Currently these logs can only be read if the users go to the CloudWatch console or use the AWS CLI. A CLI command part of the Ribble CLI would make it easier for users to collect the logs.
4. **Configuration:** the costs of a job are directly linked to the configuration used for the Lambda functions and this may not always be optimal. For example, running jobs with Lambdas configured with 10 GB of memory for jobs that run with 128 MB would incur unnecessary costs. For recurrent jobs, it would be ideal if Ribble could learn how to configure the job given its previous experience running with similar workloads.
5. **Chaining jobs:** MapReduce allows for the output of a job to be part of the input job. This allows the users to define a wider range of jobs.
6. **Testing:** overall, more unit and functional testing would increase Ribble's quality.

9 References

- Bergamaschi, S. *et al.* (2017) 'BigBench Workload Executed by using Apache Flink', *Procedia Manufacturing*, 11(June), pp. 695–702. doi: 10.1016/j.promfg.2017.07.169.
- Carbone and Asterios Katsifodimos and Stephan Ewen and Volker Markl and Seif Haridi and Kostas Tzoumas (2016) 'Apache Flink: Stream and Batch Processing in a Single Engine Paris', *Undefined*, (October 2016), pp. 28–38.
- Dean, J. and Ghemawat, S. (2004) 'MapReduce: Simplified data processing on large clusters', *OSDI 2004 - 6th Symposium on Operating Systems Design and Implementation*, pp. 137–149. doi: 10.21276/ijre.2018.5.5.4.
- Ghazi, M. R. and Gangodkar, D. (2015) 'Hadoop, mapreduce and HDFS: A developers perspective', *Procedia Computer Science*, 48(C), pp. 45–50. doi: 10.1016/j.procs.2015.04.108.
- Giménez-Alventosa, V., Moltó, G. and Caballer, M. (2019) 'A framework and a performance assessment for serverless MapReduce on AWS Lambda', *Future Generation Computer Systems*, 97(March), pp. 259–274. doi: 10.1016/j.future.2019.02.057.
- Glushkova, D., Jovanovic, P. and Abelló, A. (2019) 'Mapreduce performance model for Hadoop 2.x', *Information Systems*, 79, pp. 32–43. doi: 10.1016/j.is.2017.11.006.
- Gurusamy, Vairaprakash. Kannan, S. Nandhini, K. (2017) 'The Real Time Big Data Processing Framework Advantages and Limitations', *International Journal of Computer Sciences and Engineering*, 5(12), pp. 305–312. doi: 10.26438/ijcse/v5i12.305312.
- Jonas, E. *et al.* (2017) 'Occupy the cloud: Distributed computing for the 99%', *SoCC 2017 - Proceedings of the 2017 Symposium on Cloud Computing*, pp. 445–451. doi: 10.1145/3127479.3128601.
- Jonas, E. *et al.* (2019) 'Cloud Programming Simplified: A Berkeley View on Serverless Computing', pp. 1–33. Available at: http://arxiv.org/abs/1902.03383.
- Kim, Y. and Lin, J. (2018) 'Serverless Data Analytics with Flint', *IEEE International Conference on Cloud Computing, CLOUD*, 2018-July, pp. 451–455. doi: 10.1109/CLOUD.2018.00063.
- Pike, R. *et al.* (2005) 'Interpreting the data: Parallel analysis with Sawzall', *Scientific Programming*, 13(4 SPEC. ISS.), pp. 277–298. doi: 10.1155/2005/962135.
- Werner, S., Girke, R. and Kuhlenkamp, J. (2020) 'An Evaluation of Serverless Data Processing Frameworks', *WOSC 2020 - Proceedings of the 2020 6th International Workshop on Serverless Computing, Part of Middleware 2020*, pp. 19–24. doi: 10.1145/3429880.3430095.

Online resources:

- https://amplab.cs.berkeley.edu/benchmark/ consulted on 10/4/2022
- https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/ consulted on 20/12/2021
- https://benjamincongdon.me/blog/2018/05/02/Introducing-Corral-A-Serverless-MapReduce-Framework/ consulted on 22/12/2021
- https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html consulted on 05/01/2022
- https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dg.pdf consulted on 05/01/2022

- https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/quotas-messages.html consulted on 05/01/2022
- https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html consulted on 20/4/2022
- https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html consulted on 05/01/2022
- https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf consulted on 05/01/2022
- https://docs.aws.amazon.com/lambda/latest/dg/golang-handler.html consulted on 05/01/2022
- https://github.com/bcongdon/corral consulted on 22/12/2021
- https://learn.lboro.ac.uk/pluginfile.php/1682291/mod_resource/content/4/lecture07-pubsubsystems.pdf consulted on 20/4/2022
- https://learn.lboro.ac.uk/pluginfile.php/1690389/mod_resource/content/24/Lecture08_FaultTolerance.pdf consulted on 20/4/2022
- https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/learn-flink/overview/ accessed on 03/01/2022
- https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/concepts/flink-architecture/ consulted on 03/01/2022
- https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/concepts/stateful-stream-processing/ consulted on 03/01/2022
- https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf consulted on 10/4/2022

10 Appendix

Instructions to run the CLI

Prerequisites

To run Ribble you need to have the following in your local machine:

- Docker
- AWS CLI installed and configured. Instructions can be found at https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html
- To setup Ribble your AWS user needs to have AdministratorAccess or permission to create roles and policies
- Go (at least version 1.16)
- Make

Download the Ribble project

```
git clone git@github.com:josenarvaezp/ribble.git
```

** NOTE: the GitHub repository may be set to private. Contact jose.narvaez.p@gmail.com to get access.

Build Ribble CLI

To build the CLI tool run:

```
make build_cli
```

This should have created an executable called 'ribble' within your current directory.

Set credentials

Ribble needs AWS permissions to access S3, SQS, Lambda, IAM, and ECR to run a processing job. To facilitate adding these permissions into your account you can use the `set-credentials` command. The `set-credentials` command creates an AWS role called `ribble` and it assigns to it the policies it requires to access the resources needed. It then gives the specified user permission to assume this role. Hence this command can be used by an administrator in the AWS account (someone with AWS AdministratorAccess) to give access to different users within the account that need to run Ribble jobs.

```
ribble set-credentials \
    --account-id <your-account-id> \
    --username <aws-username>
```

Use --local to create the credentials in localstack.

Build

The `build` command is used to create the resources locally that are needed to run the job. Specifically it auto-generates AWS lambda `.go` files for the job coordinator, mapper and reducer functions. It also auto-generates `Dockerfiles` for each of them and builds the images.

```
ribble build --job <path-to-your-job-definition>
```

Output:

```
Generating resources...
Building docker images...
Build successful with Job ID:  308866c6-2ef0-4f80-868e-6b1760da8eb9
```

Upload

The `upload` command is used to upload all the resources that were genererated by the `build` command and creates additional resources needed to run the Ribble job such as the SQS queues that hold the intermediate data, a bucket for the job, amongst other.

```
ribble upload --job-id <id-of-job>
```

Output:

```
Creating resources...
Creating job S3 bucket...
Generating mappings...
Writing mappings to S3...
Creating streams in SQS...
Creating log stream in CloudWatch...
Creating SQS dead-letter queue...
Uploading Lambda functions...
Upload successful with Job ID:  308866c6-2ef0-4f80-868e-6b1760da8eb9
```

Run

The `run` command is used to run the job with the given job id. Note that this command runs the Ribble job but does not wait until it has completed. If any errors occurred or you want to know the status of the job, you need to use the `track` command.

```
ribble run --job-id <id-of-job>
```

Output:

```
Running job:  308866c6-2ef0-4f80-868e-6b1760da8eb9
```

Track

The `track` command is used to track the progress of a job. It can tell you how many mappers and reducers are left in the job or if the job has been completed.

```
ribble track --job-id <id-of-job>
```

Output:

```
INFO[0000] Coordinator starting...                       Timestamp="54305-01-28
13:11:09 +0000 GMT"
INFO[0000] Waiting for 1 mappers...                      Timestamp="54305-01-28
13:11:09 +0000 GMT"
INFO[0000] Mappers execution completed...                Timestamp="54305-01-28
14:06:08 +0000 GMT"
INFO[0000] Waiting for 1 reducers...                     Timestamp="54305-01-28
14:06:08 +0000 GMT"
INFO[0000] Reducers execution completed...               Timestamp="54305-01-28
14:58:52 +0000 GMT"
INFO[0000] Waiting for final reducer...                  Timestamp="54305-01-28
14:58:52 +0000 GMT"
INFO[0000] Final reducer execution completed...          Timestamp="54305-01-28
16:23:16 +0000 GMT"
INFO[0000] Job completed successfully, output is available at the S3 bucket 308866c6-
2ef0-4f80-868e-6b1760da8eb9...  Timestamp="54305-01-28 16:23:16 +0000 GMT"
```

Local testing

For local testing you can use Localstack, a docker service that replicates AWS locally. You can either use the AWS CLI by using the `--endpoint-url` flag like: `aws --endpoint-url=http://localhost:4566 s3 ls` or you can download awslocal at https://github.com/localstack/awscli-local.

To start localstack run:

```
docker-compose up -d
```

Once localstack is running, create a bucket and upload files:

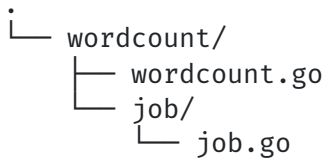
```
awslocal s3 mb s3://my-input-bucket
awslocal s3 cp test.txt  s3://my-input-bucket/test.txt
```

When setting up the job, remember to set `Local` to true in the configuration. You can then use the Ribble CLI as described in the first section. While you can test the `set-credentials` command by setting the `--local` flag to true, note that localstack does not check IAM roles or users so you can skip that.

Once the jobs finish you will be able to get the result from S3 using the awslocal CLI.

A full word count example

Recall that to define a job you need to have a file where you define your map, filter and sort functions and a main file where you configure the job. The following example follow the given structure:



Contents of wordcount.go:

```go
package wordcount

import (
    "bufio"
    "log"
    "os"
    "sort"
    "strings"

    "github.com/josenarvaezp/displ/pkg/aggregators"
)

func WordCount(filename string) aggregators.MapAggregator {
    file, err := os.Open(filename)
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()

    scanner := bufio.NewScanner(file)

    // initialize map
    output := aggregators.NewMap()

    for scanner.Scan() {
        line := scanner.Text()
        words := strings.Fields(line)
        for _, word := range words {
            output.AddSum(word, 1)
        }
    }

    return output
}
```

```
// Having filters the words that have less than 5 in their sum
func Having(mapAggregator aggregators.MapAggregator)
aggregators.MapAggregator {
    // delete all items from map that have less then 5 count
    for key, aggregator := range mapAggregator {
        if aggregator.ToNum() < 5 {
            delete(mapAggregator, key)
        }
    }

    return mapAggregator
}

type AggregatorPairList []aggregators.AggregatorPair

func (p AggregatorPairList) Len() int      { return len(p) }
func (p AggregatorPairList) Swap(i, j int) { p[i], p[j] = p[j], p[i] }
func (p AggregatorPairList) Less(i, j int) bool {
    return p[i].Value < p[j].Value
}

// Sort sorts the output by value in ascending order
func Sort(ma aggregators.MapAggregator) sort.Interface {
    keys := make(AggregatorPairList, len(ma))
    i := 0
    for k, v := range ma {
        keys[i] = aggregators.AggregatorPair{Key: k, Value: v.ToNum()}
        i++
    }

    sort.Sort(keys)

    return keys
}
```

Contents of job.go:

```
package main

import (
    "github.com/josenarvaezp/displ/examples/wordcount"
    "github.com/josenarvaezp/displ/pkg/ribble"
)

func main() {
    // define job's config
    config := ribble.Config{
        InputBuckets:        []string{"my-input-bucket"},
        Region:              "eu-west-2",
        Local:               true,
        LogLevel:            1,
        AccountID:           "000000000000",
        Username:            "my-iam-user",
        LogicalSplit:        true,
        RandomizedPartition: false,
    }

    // define job
    ribble.Job(
        wordcount.WordCount,
        wordcount.Having,
        wordcount.Sort,
        config,
    )
}
```

Auto-generated Lambda Go code

The map Lambda function:

```
// Code generated by ribble DO NOT EDIT.
// |\   \\\\__     o
// | \_/    o \    o
// > _   (( <_  oo
// | / \__+___/
// |/     |/

package main

import (
    "context"
    "os"

    "github.com/aws/aws-lambda-go/lambda"
    log "github.com/sirupsen/logrus"

    "github.com/josenarvaezp/displ/pkg/lambdas"
    "github.com/josenarvaezp/displ/examples/wordcount"
)

var m *lambdas.Mapper

func init() {
    // set logger
    log.SetLevel(log.ErrorLevel)

    var err error
    m, err = lambdas.NewMapper(true)
    if err != nil {
        log.WithError(err).Fatal("Error starting mapper")
        return
    }
}

func HandleRequest(ctx context.Context, request lambdas.MapperInput) error
{
    // update mapper
    m.UpdateMapperWithRequest(ctx, request)

    // set mapper logger
    mapperLogger := log.WithFields(log.Fields{
        "Job ID": m.JobID.String(),
        "Map ID": m.MapID.String(),
    })

    // keep a dictionary with the number of batches per queue
    batchMetadata := make(map[int]int64)
```

```
    for _, object := range request.Mapping.Objects {
        // download file
        filename, err := m.DownloadFile(object)
        if err != nil {
            mapperLogger.
                WithFields(log.Fields{
                    "Bucket": object.Bucket,
                    "Object": object.Key,
                }).
                WithError(err).
                Error("Error downloading file")
            return err
        }

        // user function starts here
        mapOutput := lambdas.RunMapAggregator(*filename,
wordcount.WordCount)

        // send output to reducers via queues
        err = m.EmitMap(ctx, mapOutput, batchMetadata)
        if err != nil {
            mapperLogger.
                WithFields(log.Fields{
                    "Bucket": object.Bucket,
                    "Object": object.Key,
                }).
                WithError(err).
                Error("Error sending map output to reducers")
            return err
        }

        // clean up file in /tmp
        err = os.Remove(*filename)
        if err != nil {
            mapperLogger.
                WithFields(log.Fields{
                    "Bucket": object.Bucket,
                    "Object": object.Key,
                }).
                WithError(err).
                Error("Error cleaning file from /temp")
            return err
        }
    }

    // send batch metadata to sqs
    if err := m.SendBatchMetadata(ctx, batchMetadata); err != nil {
        mapperLogger.WithError(err).Error("Error sending metadata to
streams")
        return err
    }

    // send event to queue indicating this mapper has completed
```

```
    if err := m.SendFinishedEvent(ctx); err != nil {
        mapperLogger.WithError(err).Error("Error sending done event to
stream")
        return err
    }

    return nil
}

func main() {
    lambda.Start(HandleRequest)
}
```

The coordinator Lambda function:

```
// Code generated by ribble DO NOT EDIT.
// |\   \\\\__     o
// | \_/    o \    o
// > _   (( <_  oo
// | / \__+___/
// |/     |/

package main

import (
    "context"
    "fmt"

    "github.com/aws/aws-lambda-go/lambda"
    log "github.com/sirupsen/logrus"

    "github.com/josenarvaezp/displ/pkg/lambdas"
)

var c *lambdas.Coordinator

func init() {
    // set logger
    log.SetLevel(log.ErrorLevel)

    var err error
    c, err = lambdas.NewCoordinator(true)
    if err != nil {
        log.WithError(err).Fatal("Error starting coordinator")
        return
    }
}

func HandleRequest(ctx context.Context, request lambdas.CoordinatorInput)
error {
    // update coordinator
```

```
    c.UpdateCoordinatorWithRequest(ctx, request)

    // set coordinator logger
    coordinatorLogger := log.WithFields(log.Fields{
        "Job ID": c.JobID.String(),
    })

    // log init
    nextLogToken, _ := c.LogEvents(
        ctx,
        []string{
            "Coordinator starting...",
            fmt.Sprintf("Waiting for %d mappers...", request.NumMappers),
        },
        nil, // empty token as it is the first log
    )

    // start mappers
    err := c.StartMappers(ctx, request.NumQueues, request.FunctionName)
    if err != nil {
        coordinatorLogger.WithError(err).Error("Error starting the
mappers")
        return err
    }

    // waits until mappers are done
    nextLogToken, err = c.AreMappersDone(ctx, nextLogToken)
    if err != nil {
        coordinatorLogger.WithError(err).Error("Error reading mappers done
queue")
        return err
    }

    // log mappers done
    nextLogToken, _ = c.LogEvents(
        ctx,
        []string{
            "Mappers execution completed...",
            fmt.Sprintf("Waiting for %d reducers...", request.NumQueues),
        },
        nextLogToken,
    )

    // invoke reducers
    if err := c.InvokeReducers(ctx, "map_aggregator"); err != nil {
        coordinatorLogger.WithError(err).Error("Error invoking reducers")
        return nil
    }

    // wait until reducers are done
    nextLogToken, err = c.AreReducersDone(ctx, nextLogToken)
    if err != nil {
```

```
        coordinatorLogger.WithError(err).Error("Error reading reducers
done queue")
        return err
    }

    // log reducers done
    nextLogToken, _ = c.LogEvents(
        ctx,
        []string{
            "Reducers execution completed...",
            fmt.Sprintf(
                "Job completed successfully, output is available at the S3
bucket %s...",
                c.JobID.String(),
            ),
        },
        nextLogToken,
    )

    // indicate reducers are done
    if err := c.WriteDoneObject(ctx, "done"); err != nil {
        coordinatorLogger.WithError(err).Error("Error writing done
signal")
        return err
    }

    return nil
}

func main() {
    lambda.Start(HandleRequest)
}
```

The reducer Lambda function:

```
// Code generated by ribble DO NOT EDIT.
// |\   \\\\__     o
// | \_/    o \    o
// > _   (( <_  oo
// | / \__+___/
// |/     |/

package main

import (
    "context"
    "encoding/json"
    "fmt"
    "strconv"
    "sync"
```

```
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-sdk-go-v2/service/sqs"
    sqsTypes "github.com/aws/aws-sdk-go-v2/service/sqs/types"
    log "github.com/sirupsen/logrus"

    "github.com/josenarvaezp/displ/pkg/lambdas"
    "github.com/josenarvaezp/displ/pkg/aggregators"


    "github.com/josenarvaezp/displ/examples/wordcount"

)

var r *lambdas.Reducer

func init() {
    // set logger
    log.SetLevel(log.ErrorLevel)

    var err error
    r, err = lambdas.NewMapReducer(true)
    if err != nil {
        log.WithError(err).Fatal("Error starting reducer")
        return
    }
}

func HandleRequest(ctx context.Context, request lambdas.ReducerInput)
error {
    // update reducer
    r.UpdateReducerWithRequest(ctx, request)

    // get reduce queue information
    queueName := fmt.Sprintf("%s-%d", r.JobID.String(),
request.QueuePartition)
    queueURL := lambdas.GetQueueURL(queueName, r.Region, r.AccountID,
r.Local)

    // set reducer logger
    reducerLogger := log.WithFields(log.Fields{
        "Job ID":          r.JobID.String(),
        "Reducer ID":      r.ReducerID.String(),
        "Queue Partition": r.QueuePartition,
    })

    // set wait group
    var wg sync.WaitGroup

    // get checkpoint data
    checkpointData, err := r.GetCheckpointData(ctx, &wg)
    if err != nil {
        reducerLogger.WithError(err).Error("Error reading checkpoint")
```

```
        return err
    }

    // batch metadata - number of batches the reducer needs to process
    totalBatchesToProcess, err := r.GetNumberOfBatchesToProcess(ctx)
    if err != nil {
        reducerLogger.WithError(err).Error("Error getting queue metadata")
        return err
    }
    totalProcessedBatches := 0

    // checkpoint info
    processedMessagesWithoutCheckpoint := 0
    checkpointData.LastCheckpoint++

    // holds the intermediate results
    intermediateReducedMap := make(aggregators.MapAggregator)

    // processedMessagesDeleteInfo holds the data to delete messages from
queue
    processedMessagesDeleteInfo :=
make([]sqsTypes.DeleteMessageBatchRequestEntry,
lambdas.MaxMessagesWithoutCheckpoint)

    // use same parameters for all get messages requests
    recieveMessageParams := &sqs.ReceiveMessageInput{
        QueueUrl:            &queueURL,
        MaxNumberOfMessages: lambdas.MaxItemsPerBatch,
        MessageAttributeNames: []string{
            lambdas.MapIDAttribute,
            lambdas.BatchIDAttribute,
            lambdas.MessageIDAttribute,
        },
        WaitTimeSeconds: int32(5),
    }

    // recieve messages until we are done processing all queue
    for true {
        if processedMessagesWithoutCheckpoint ==
lambdas.MaxMessagesBeforeCheckpointComplete &&
checkpointData.LastCheckpoint != 1 {
            // check that the last checkpoint has completed before
processing any more messages
            // we give a buffer of 15,000 new messages for saving the
checkpoint which happens
            // in the background. If this point is reached it means we
have processed 115,000 messages
            // without deleting from the queue which is close to the aws
limit for queues
            wg.Wait()
        }
```

```
        if processedMessagesWithoutCheckpoint ==
lambdas.MaxMessagesWithoutCheckpoint {
            // We need to delete the messages read from the sqs queue and
we create a checkpoint
            // in S3 as the fault tolerant mechanism. Saving the
checkpoint can be done concurrently
            // in the background while we keep processing messages

            // merge the dedupe map so that the read dedupe map is up to
date
            wg.Add(1)
            go r.Dedupe.Merge()

            // save intermediate dedupe
            wg.Add(1)
            go r.SaveIntermediateDedupe(ctx,
checkpointData.LastCheckpoint, r.Dedupe.WriteMap, &wg)

            // save intermediate map
            wg.Add(1)
            go r.SaveIntermediateOutput(ctx, intermediateReducedMap,
checkpointData.LastCheckpoint, &wg)

            // update output map with reduced intermediate results
            wg.Add(1)
            go r.Output.UpdateOutput(intermediateReducedMap, &wg)

            // delete all messages from queue
            wg.Add(1)
            go r.DeleteIntermediateMessagesFromQueue(ctx, queueURL,
processedMessagesDeleteInfo, &wg)

            // update checkpoint info
            checkpointData.LastCheckpoint++
            processedMessagesWithoutCheckpoint = 0
            processedMessagesDeleteInfo =
make([]sqsTypes.DeleteMessageBatchRequestEntry,
lambdas.MaxMessagesWithoutCheckpoint)
            intermediateReducedMap = make(aggregators.MapAggregator)
            r.Dedupe.WriteMap = lambdas.InitDedupeMap()
        }

        // call sqs receive messages
        output, err := r.QueuesAPI.ReceiveMessage(ctx,
recieveMessageParams)
        if err != nil {
            reducerLogger.WithError(err).Error("Error reading from queue")
            return err
        }

        // process messages
        for _, message := range output.Messages {
            processedMessagesWithoutCheckpoint++
```

```
            // add delete info

processedMessagesDeleteInfo[processedMessagesWithoutCheckpoint] =
sqsTypes.DeleteMessageBatchRequestEntry{
                Id:            message.MessageId,
                ReceiptHandle: message.ReceiptHandle,
            }

            // get message attributes
            currentMapID :=
message.MessageAttributes[lambdas.MapIDAttribute].StringValue
            currentBatchID, err :=
strconv.Atoi(*message.MessageAttributes[lambdas.BatchIDAttribute].StringVa
lue)
            if err != nil {
                reducerLogger.WithError(err).Error("Error getting message
batch ID")
                return err
            }
            currentMessageID, err :=
strconv.Atoi(*message.MessageAttributes[lambdas.MessageIDAttribute].String
Value)
            if err != nil {
                reducerLogger.WithError(err).Error("Error getting message
ID")
                return err
            }

            // check if message has already been processed
            if exists := r.Dedupe.BatchExists(*currentMapID,
currentBatchID); exists {
                if r.Dedupe.IsBatchComplete(*currentMapID, currentBatchID)
{
                    // ignore as it is a duplicated message
                    continue
                }

                if r.Dedupe.IsMessageProcessed(*currentMapID,
currentBatchID, currentMessageID) {
                    // ignore as it is a duplicated message
                    continue
                }

                // message has not been processed
                // add processed message to dedupe map
                r.Dedupe.UpdateMessageProcessed(*currentMapID,
currentBatchID, currentMessageID)

                // check if we are done processing batch from map
                if r.Dedupe.IsBatchComplete(*currentMapID, currentBatchID)
{
                    totalProcessedBatches++
```

```
                    // delete processed map from dedupe
                    r.Dedupe.DeletedProcessedMessages(*currentMapID,
currentBatchID)
                }
            } else {
                // no messages for batch have been processed - init dedupe
data for batch
                r.Dedupe.InitDedupeBatch(*currentMapID, currentBatchID,
currentMessageID)
            }

            // process message
            // unmarshall message body
            var reduceMessage *aggregators.ReduceMessage
            body := []byte(*message.Body)
            err = json.Unmarshal(body, &reduceMessage)
            if err != nil {
                return err
            }

            if err := intermediateReducedMap.Reduce(reduceMessage); err !=
nil {
                reducerLogger.WithError(err).Error("Error processing
message")
                return err
            }
        }

        // check if we are done processing values
        if totalProcessedBatches == *totalBatchesToProcess {
            break
        }
    }

    // wait in case reducers is saving checkpoint in the background
    wg.Wait()

    // update output map with reduced intermediate results
    wg.Add(1)
    go r.Output.UpdateOutput(intermediateReducedMap, &wg)
    wg.Wait()


    // filter results
    r.Output = lambdas.RunFilter(r.Output, wordcount.Having)


    // generate key for output
    key := fmt.Sprintf("output/%s", r.ReducerID.String())


    // sort output
    sortedOutput := lambdas.RunSort(r.Output, wordcount.Sort)
```

```
    // write sorted reducer output
    err = r.WriteSortedReducerOutput(ctx, sortedOutput, key)
    if err != nil {
        reducerLogger.WithError(err).Error("Error writing reducer output")
        return err
    }


    // delete all messages from queue
    wg.Add(1)
    go r.DeleteIntermediateMessagesFromQueue(ctx, queueURL,
processedMessagesDeleteInfo, &wg)
    wg.Wait()

    // indicate reducer has finished
    err = r.SendFinishedEvent(ctx)
    if err != nil {
        reducerLogger.WithError(err).Error("Error sending done message")
        return err
    }

    return nil
}

func main() {
    lambda.Start(HandleRequest)
}
```

Figure 14: example of a unit test function using mocks

```
func Test_GetNumberOfBatchesToProcess_HappyPath(t *testing.T) {
    ctx := context.Background()
    jobID := uuid.New()
    queueUrl := fmt.Sprintf(
        "https://sqs.eu-west-2.amazonaws.com/000000000000/%s-1-meta",
        jobID.String(),
    )

    // set expectations
    expectedInput := &sqs.ReceiveMessageInput{
        QueueUrl:            &queueUrl,
        MaxNumberOfMessages: 10,
        WaitTimeSeconds:     5,
    }

    expectedOutput1 := &sqs.ReceiveMessageOutput{
        Messages: []types.Message{
            {
                Body: getExpectedMessage(t),
            },
        },
    }

    expectedOutput2 := &sqs.ReceiveMessageOutput{
        Messages: []types.Message{
            {
                Body: getExpectedMessage(t),
            },
        },
    }
```

```
    // create mock
    sqsMock := new(mocks.QueuesAPI)
    // simulate a repeated message - one should be ignored
    sqsMock.On("ReceiveMessage", ctx, expectedInput).Return(expectedOutput1,
nil).Twice()
    // simulate another message
    sqsMock.On("ReceiveMessage", ctx, expectedInput).Return(expectedOutput2,
nil).Once()

    reducer := lambdas.Reducer{
        JobID:          jobID,
        Region:         "eu-west-2",
        AccountID:      "000000000000",
        Local:          false,
        NumMappers:     2,
        QueuePartition: 1,
        QueuesAPI:      sqsMock,
    }

    numBatches, err := reducer.GetNumberOfBatchesToProcess(ctx)
    require.Nil(t, err)
    // assert numBatches is 10 as 2 mappers were used
    // each indicating they sent 5 messages
    assert.Equal(t, 10, *numBatches)
}
```