Serverless Distributed Data Processing

Daniel Bin Schmid, Ermias Habtegabr, Kristiyan Sakalyan

February 2023

1 Data Processing Pipeline Design

In fig. 1 the high level system diagram of our Azure Functions App can be seen. The grey marked area denotes the scope of the Azure Function Application. The trigger of the data analytics pipeline is designed to be a HTTP POST request with a filelist inside a JSON file as request body due to the reason, that it allows for more efficient batch processing instead of processing incoming files only sequentially. Furthermore, unlike when triggering the pipeline with e.g. a Azure Blob Trigger, the data must not necessarily be uploaded to a data storage of the pipeline. Instead, users can upload the pipeline to any accessible location, and the Orchestrator can forward location and metadata of the files to the Azure Functions App. The filelist inside the HTTP POST request has no restrictions in terms of number of files but can also just contain the filename of a single file.

1.1 Partitioning, Parallelization, and Merging

The first function of the pipeline is the so-called 'Partitioner', who receives a JSON file containing the filenames to be processed. It sends the work to multiple workers, who are instances of a worker Azure Function. We found that it suffices to instantiate each worker with one whole file instead of further subpartitioning the files. Thus, if N_{files} is the number of files inside the filelist, N_{files} merger Azure Functions will be instantiated. The input of the worker Azure Function is the filename of the file to be processed and a location where to stream the results to. The results are streamed to 'Aggregation', which is issued by the Partitioner. The Partitioner gives the Aggregation Azure Function knowledge about the identities of the partitiones. Therefore, the Aggregation Azure Function gains knowledge about what results to expect and can wait for all tasks to be completed. Note, that if the Partitioner wouldn't receive a batch of files but a sequential stream of single files, the problem would occur about how the Aggregation Azure Function would know for what partitions it is waiting for. Finally, the Aggregation Azure Function forwards the aggregated result to the 'Merger', who loads the current result from a Result Blob Container and updates it with the freshly computed result.

2 Scaling to Large Amounts of Data

We define large amounts of data by a consistent file size of maximal 500mb and a high number of concurrent data uploads. The degree of concurrency of the data uploads is assumed to be unbounded. For our pipeline, large amounts of data thus refers to big work batches (many filenames inside a filelist POST request work batch) and the processing of potentially many of such work-batches simultanuously. Our pipeline scales with both variables, batch size (s_{batch}) and number of batches $n_{batches}$, by leveraging the power of Azure Functions, i.e. to spawn as many functions as there are files inside a file list or number of batches, respectively. However, since the Merger must coordinate the update of results in the Results Blob container and eliminate race conditions, it may become a bottleneck if $n_{batches}$ is very high and the computational costs of the work batches is uniform, as it only can merge one result at a time. The results of a single large batch, however, can be merged without overhead since the *Aggregation* can merge multiple partition results at a time. On the other hand, *Aggregation* suffers if the computational cost of the partitioned tasks are not evenly distributed since it waits for the slowest worker.

3 Sharing State Between Pipeline Stages

We implemented two pipelines, one where the states are shared with Azure Blobs and one where the states are shared with Azure Queue Storage. States to be shared are the result of the Worker Azure Function instances and

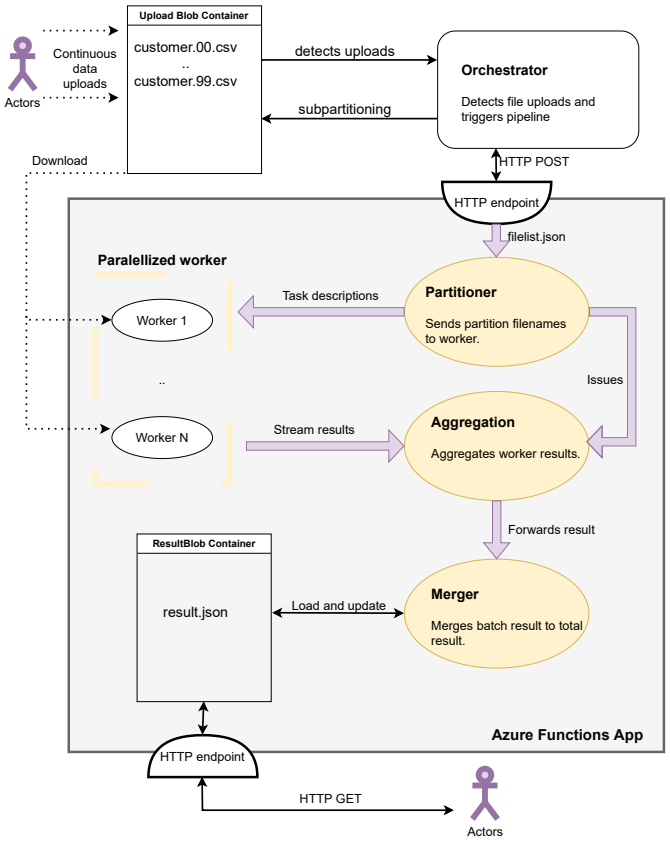

Figure 1: High level system diagram of Azure Functions App.

the Aggregation Azure Function, and data which the Azure Functions take as input. The latter, i.e. the input to the functions, can be passed with the function trigger. To be consistent with both pipelines and to have a more clear comparison, we either used only Blob Storage or only Queue Storage for both triggering the Azure Functions in-between the pipeline stages and the stream of results from the Workers. Only for the start and end of the pipeline we consistently used HTTP POST/ GET requests and Blob storage for the results.

3.1 Scaling and scale tests

We ran both of our pipelines using the same files as input and measured our performance. In order to exclude the uploading time from the measurements, we pre-upload the files and only then trigger the pipeline. In order to see how the maximum number of instances that we set influences the runtimes, we did four different executions with 5, 50, 100, and 200 as the number of maximum instances. Here are the results:

N	Blob Storage Runtime	Queue Storage Runtime
200	167s	164s
100	244s	179s
50	242s	160s
5	258s	219s

We can clearly see that increasing the number of maximum instances would influence the runtime in a positive manner - runtime improves. This happens because more instances can process the data simultaneously resulting in faster execution. **Manual Scaling** is in general bad for our pipeline because it bounds us to a maximum number of functions and we cannot predict how much data the user would upload. **Auto-Scaling** would be a good approach in terms of performance but can increase the cost of our pipelines in the scenario when we spawn more than thousands of instances. **Consumption Plan** with a maximum number of instances provides a good balance between performance and costs, as it scales in when demand is low and scales out when demand is high and is bound to N functions running in parallel.

4 Pipeline Comparison

After comparing the different pipelines with different number of batches as well as different number of instances, we observed different results in term of speed, memory usage, and consumption price. These topics will be discussed in this section.

4.1 Data processing speed and load tests

To compare the data processing speed, we benchmarked the query execution time by different means of number of batches $n_{batches}$ for a data workload of 5GB. We split the 5GB of data into 500 equal sized chunk to simulate the output of the *Orchestrator*. Furthermore, we compare the runtime when deployed on Azure against when deployed locally. In the following table our results can be seen:

Deployment type	$n_{batches}$	Queue pipeline runtime	Blob pipeline runtime
Azure	1	184s	157s
Azure	5	137s	153s
Azure	10	102s	149s
Azure	20	132s	147s
Azure	50	145s	139s
Azure	100	159s	128s
Azure	250	167s	125s
Local	Average	366.33s	780.5s

Evidently, it can be observed, that the aggregation bottleneck has a significant effect in the blob-based pipeline since there is a clear linear relationship - the execution time decreases with increasing number of batches, hence decreasing batch size. We theorize, that since blob uploads strongly depend on the network transmission rate which has high variance, the tasks completion time of the workers are non-uniform. Due to the higher variance in execution time of the workers, the merger is not a bottleneck anymore, wherefore increasing the number of batches is not hurting the performance.
In the Queue-based pipeline, the Merger bottleneck is strongly present in form of increased execution time for very large batch sizes. With too high batch size, on the other hand, also a strong Aggregation bottleneck is visible. We found, that a trade-off in number of batches and batch size with 10 number of batches, thus a batch size of 50 is optimal. In conclusion, the queue pipeline is better suitable for the tasks if the batch size is chosen appropriately. The blob pipeline introduces randomness in terms of termination time of a single worker but handles, generally speaking, large data amounts slightly worse.

4.2 Storage

In terms of Queue Storage, for a standard query of a batch size of 50, our pipeline has a message count of $n_{tasks} \cdot 2 + n_{stages} = 50 \cdot 2 + 3 = 103$. Therefore, for a single standard query, the queue costs are given by[1] $n_{messages} \cdot 0.0004\$ \cdot \frac{1}{10.000} = 4.12\dot{1}0^{-6}\$$. For the Blob-based pipeline we have 103 number of writes and 103 number of blob reads, analogously computed. This gives a cost estimation of[2] $103 \cdot \frac{1}{10.000} \cdot 0.065\$ + 103 \cdot \frac{1}{10.000} \cdot 0.005\$ = 7.21 \cdot 10^{-4}\$$ per query. Hence, the Queue-storage based pipeline is cheaper by a magnitude of 2 in terms of storage costs for shari.

4.3 Price

In order to estimate the cost we followed the price calculations displayed here: Azure Function Billing Calculation. We ran our pipelines multiple times on the partitioned "customer.csv" file that you provided us with and gained insights about the average execution time of each function, memory consumption, and the number of executions per pipeline. This allowed us to calculate a price per execution, resource consumption price, and total price for both pipelines.
For a single query for the blob-based pipeline, we observed a resource consumption of 1.2GB and a summed up execution time of 686s (summed up execution time of all Azure Functions, excluding parallelism). In total, this yields $823.3GBs$, which results in 0.012348 Euro costs per query by taking into account the resource consumption

[1] `https://azure.microsoft.com/de-de/pricing/details/storage/queues/`
[2] `https://azure.microsoft.com/de-de/pricing/details/storage/blobs/`

price of $0.000015 \frac{Euro}{GBs}$. Analogously, the Queue-based pipeline costs 0.012546 per query. Therefore, the Blob-based pipeline is cheaper by a factor of 1.6%
*The price per execution for processing 100 files n times is actually scaled up with 10^3 because it is very small.

As you can see in the figure above, initially the price deviation between both pipelines is little but becomes bigger as the number of execution grows. In conclusion, the pipeline using Azure Queue is more expensive than the one using Blob Storage.

5 Changes for Instance Failures - Fault Tolerance

The Queue and Blob Pipeline are both fault-tolerant to some extent. The Azure Function framework handles failures of Function instances in the following way:

1. In the case of a blob-triggered function, if the function fails, the framework writes a poison blob to a queue (created by Azure) on the storage account specified by the connection. In the case of a queue-triggered function, if the function fails, a timeout is triggered automatically by the Azure Framework, and the task is enqueued again to the tasks queue.

2. Then the function instance is spawned with the same arguments.

Steps 1 and 2 are then repeated five times by default (This value is of course configurable but we decided that 5 retries are reasonable for the Blob and Queue Pipeline). This means that the blob pipeline is fault-tolerant if it manages to execute the function successfully with less than or equal to six tries in total. If we want to make our pipeline completely fault-tolerant, we have to remove the maximum number of retries.

5.1 Merger Fault Tolerance

There is only one scenario where the pipelines cannot recover correctly after a merge node failure. If there is a previous merge result, the merger reads it and merges it with the result from the aggregation. This means that it rewrites the file. If the node fails during writing, the function will be spawned again but the file will be corrupted. To deal with that case, we need to create a replicated file and use it to compare both of the files. If there is a difference, this would imply that the file has been corrupted. Thus we can use the replicated file, which is only updated on successful merges.

5.2 Aggregation Fault Tolerance

Blob Pipeline. In the blob pipeline we have aggregation fault-tolerance because the aggregation functions check if all of the results have been produced and only then issue merging jobs. In order to do that, they scan the containers and check if all partitions of the file are there. Since the container files can't change, even if the function fails, when it is instantiated again, it will produce the same result.
Queue Pipeline We can also make the queue pipeline fault-tolerant if each state change of the aggregation is stored in external storage.

5.3 Worker Fault Tolerance

Both pipelines are already fault-tolerant to worker failures if the task is successfully executed within six tries.

6 Main challenges

The main challenge was to synchronise reading and writing to shared data/ shared states. As the pipeline has a high degree of concurrency, we often faced race conditions.