universität innsbruck

Leopold–Franzens–University
Austria, Innsbruck

Institute of Computer Science
Distributed and Parallel Systems Group

Distributed Systems Proseminar 2020W

TITLE:
Project 3 - Stock Prices

Project Report

Supervisor: Fedor Smirnov, Sashko Ristov

TEAM: Heine Maximilian,
Kotrba Stefan, Larcher Thomas

Innsbruck
31 January 2021

Abstract

In this project we have created a workflow to be executed in the terminal with a simple user data input. The input gets processed from a function and then executed in parallel for every input you have given. At the end you get a chart with the already existing data, the interpretation of the algorithm and with a prediction for the future.

In the parallel execution of the workflow we work with the S3. After we processed data, create new data or create the charts we always store the result in a S3 bucket. Every point in that list is a independent function. These functions are written in Python and use: Boto3 for the communication with the S3. Alpha Vantage to get the data accordingly to you input. And we use QuickChart for creating the final output charts.

A program should always be efficient and not cost to much. Therefor we created the parallel part. This saves us time for the execution of multiple input variables at the same time. But such a project comes also with some difficulties and limitations. Such a limitation would be the student account, which only allows us to execute five input parameter at once.
We had trouble finding the right algorithm and file system for that job. This is further explained in 3.

Contents

1 Introduction **1**

2 Solution **2**
2.1 Helper 2
2.2 Parallel For 3
2.3 Retrieve Stock Prices 3
2.4 Predict 3
2.5 Visualize 3
2.6 Execution 3

3 Challenges **5**

4 Evaluation **6**
4.1 Experiment 1 (Cold Start vs Warm Start) 6
4.2 Experiment 2 (Input sizes and workflow overhead) 6

5 Conclusion (and future work) **8**
5.1 Future work 8
5.2 Recommendations for the PS in the future 8

1 Introduction

Our Goal with this project was to develop a simple workflow, where a user can easily get past and predicted future prices from stocks of his choosing. We focused on keeping the workflow as well as the functions as simple as possible while still maintaining good performance.
Similarly we wanted to keep it simple for the user. To run a workflow, a user simply needs to provide the stock-symbols he is looking for and the name of his s3 bucket, where the predictions will be stored. At the end of the workflow it returns the names of the images inside the bucket to the user, so he can easily find them.
The distributed workflow speeds up the task significantly as soon as you request more than one stock. A user can retrieve stock predictions of multiple symbols in almost the same time as he can with only one. This is a huge benefit over a serial approach. We distribute the workflow so each symbol can be processed on his own, as discussed later in this report.
Deploying the FC on a distributed system - or in our case especially on AWS - made a lot of sense, because we could make use of different services like S3, EFS and VPC, which all work together nicely and can be used by the lambda functions.

2 Solution

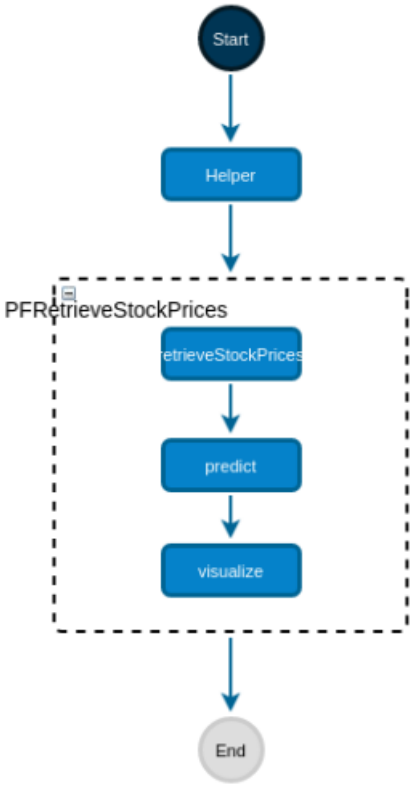

Figure 2.1: AFCL Workflow

The workflow, seen in Figure 2.1, can be divided up in 5 pieces, which are described from section 2.1 to 2.5. The input for the Terminal execution is a Json file which contains a list of all the wanted predictions(symbols) and the bucked name. The output is a collection of names from the created charts(one chart per symbol). A chart can be downloaded form the S3 when needed.

2.1 Helper

The Helper is a function to use a minimized and user friendly input and create an output, which can be used for the parallel for. The input is a Json file with the symbols in an

array and the name of the bucket. The output is the same symbols, the length of the array and the bucket name stretched to the length. The complexity of this function is O(n).

2.2 Parallel For

In the parallel for we execute the other functions parallel. It executes the subworkflow for every symbol in the list. The input is the modified input throw the helper and the output is the the forwarded output from the visualize function.
With the parallelization we can execute up to 5 symbols at the same time (restriction from the API (student accounts)) without having any changes in the execution time. Without the parallel execution the you would have a longer execution time. Further optimisation is explained in 4. Because the input of the functions 2.3 - 2.5 are two strings these functions have a complexity of O(1).

2.3 Retrieve Stock Prices

Its input is one symbol and one bucked name. It uses this information to retrieve the data, which belongs to that symbol. The received data form the Alpha Vantage API request is not passed on it is saved in S3 under the bucked name it was given. The save process is handled by Boto3, which is a AWS SDK for Python. The output contains of the same symbol and the bucked name.

2.4 Predict

The data input and output is the same as in 2.3. The previous saved data in S3 is now retrieved again with Boto3. On this data a forecast algorithm gets executed which gives us a prediction of the data for future timestamps. This data is again saved in S3.

2.5 Visualize

The data input is the same as in 2.3. The output is the name of the created chart. First the data from 2.3 and 2.4 is read from the S3. From that data a chart is created and again saved in S3. The communication with S3 is handled by Boto3. The chart is created with QuickChart, which is an API for creating charts.

2.6 Execution

To execute the workflow. Open the terminal and move to /workflow in the project folder. Then type the following command:

```
java -jar xAFCL.jar StockPrediction.yaml input.json
```

The input.json is the input file for the workflow. With the symbols in an array and the bucket name.
The other two parameters are for the execution with AFCL. Stockprediction.yaml is the workflow and was created from the AFCL-Toolkit. xAFCL.jar was given.

3 Challenges

The biggest challenge we had was with the prediction. Our first attempt was to use AWS Forecast, but the training took too long and exceeded the execution time limit of 15 min for lambda functions. Our next attempt was to use fbprophet library, which worked fine, but the library was too big and exceeded the limit of 250MB for the lambda deployment package. The solution we came up with was to use AWS Elastic Filesystem. We created a filesystem, mounted it inside an EC2 instance. Then we connected via SSH with the instance and installed all the necessary libraries inside the filesystem. Finally we mounted the filesystem inside our lambda functions where needed and added the following line before the import statements in each lambda function: sys.path.append("/mnt/lambda"). Now we were able to use the library with lambda. The advantage of this approach is that it is an universal solution, the filesystem can be mounted where ever it is needed. Another advantage is that the filesystem is elastic, which means that it does not have a size limit and scales automatically. The user only have to pay what he is actually using. A disadvantages with this approach is, that it increases the lambda initialization time, because the filesystem has to be mounted before the library can be used.

4 Evaluation

In the following section the workflow performance is benchmarked by two experiments. In the first experiment we compare the effects of cold and warm start. In the second experiement we compare the execution time for different input sizes.

4.1 Experiment 1 (Cold Start vs Warm Start)

There are two steps when a lambda is launched. First one is the lambda initialization and the second one the actual execution. Warm-start means that the lambda was launched recently (last 5 minutes) and therefore is already initialized, while cold-start means that it has to be initialized before execution. The execution time for each function was measured 20 times for cold-start and warm-start and the then min/max and average was calculated.

Name	**Cold/Warm**	**Min**	**Max**	**Average**
Helper	Cold	1989ms	2312ms	2133ms
RetrieveStockPrices	Cold	7016ms	7924ms	7516ms
Predict	Cold	20663ms	23790ms	22558ms
Visualize	Cold	8253ms	9564ms	9008ms
Helper	Warm	1105ms	1623ms	1400ms
RetrieveStockPrices	Warm	545ms	1398ms	1095ms
Predict	Warm	3227ms	3530ms	3400ms
Visualize	Warm	1753ms	2470ms	1953ms

We can see that the difference between cold start and warm start is significantly lower for helper than for the others. This comes from the fact that the filesystem is mounted in all the other functions except helper.

4.2 Experiment 2 (Input sizes and workflow overhead)

In the following section the input size denotes the number of symbols passed as input to the workflow.

The helper function is always executed once, independent from the input size. The execution time of this function increases linearly with the input size, but since the computation is quite cheap, the execution time is dominated by communication and

function initialization time and not by the computation and therefore it is nearly constant for inputs between 1 and 5.
The other three functions encapsulated in a parallel-for-loop, so they are executed in parallel for each symbol. Therefore the critical path of the second part does not increase for bigger input sizes, because it is executed 100% in parallel.

Nevertheless there is some workflow overhead which increases for higher input sizes. There are two reasons for this. First, the input is distributed to the lambda functions at the start of the parallel for and and the results are collected together at the end of the parallel for. This overhead is increasing for bigger input sizes. Second, there is some communication overhead which occurs because the lambda functions can be executed on different VMs and have to exchange data.

The average execution time and the average workflow overhead were calculated for input lengths from 1 to 5. The experiment was repeated 10 times with warm start.

The workflow overhead was calculated with the following formula:

$$overhead(wf) = exec\_time(wf) - \sum_{n=1}^{n} exec\_time(f_i)$$

Input Size	**Average Execution Time**	**Average Overhead**
1	7744ms	418ms
2	8227ms	365ms
3	8151ms	398ms
4	8732ms	426ms
5	8673ms	431ms

5 Conclusion (and future work)

In this project we created a simple workflow for a user, who wants an easy way to check predictions of certain stocks. A user needs to provide the workflow with the stock symbols he is interested in and the name of his S3-bucket, which is used to store the predictions and the generated graphs. The workflow offers an all in one solution for the user. He doesn't need to check past stock prices, train a model, run predictions on this model and then visualize the results. All of these tasks are taken care of by the workflow. One can input the symbols (and the bucket name) and gets the finished graphs returned back. Furthermore this does not only work efficiently with a single symbol but also with multiple symbols. Due to the distributed approach to this problem, each symbol gets processed in parallel, so you can obtain almost no time difference between one and multiple symbols. The only real difference in execution time is between cold starts and warm starts. So running this system multiple times definitely speeds up the task.

5.1 Future work

One of the main improvements we can think of is better understanding the prediction engine used and which parameters to tweak to get a better model and hence a better prediction.
Another point to improve, if the workflow gets developed further, is to detach it from a users account and maybe manage it on a central account. In that case not every user has to go through the setup steps necessary (especially getting fbprophet to run on AWS). This would also mean, that either a new function has to be introduced or the visualize-function has to be extended to handle a download of the generated graphs or alternatively upload them to a server where a user can reach them and return the user the URLs. A last point of improvement also concerns fbprophet. In our current implementation we used EFS to host the library and we think mounting the EFS into the lambda function introduces a lot of overhead. Finding a better solution to import this library should speed up the workflow noticeably.

5.2 Recommendations for the PS in the future

The proseminar was really interesting as it was very practical and gave a good sense how we could use the given examples in a real world scenario. Running docker containers or booting up VMs on a cloud server is something I can see myself doing in the future. The course showed up well, how to do this. Regarding the project, it was an interesting task to come up with a workflow, code the needed functions and then putting in all together

into what seems like one single program. Using everything we learned in the course up to this point it was a challenging but fun task to do. So i would definitely keep the project part of the course.

I think it is good to focus on only one cloud provider. AWS seems like a good choice for that because it offers a wide variety of services and has good documentation on how to use those services. If you encounter a problem, you can't solve by reading the docs, there is also a big community and you can always find answers to your problem online. Also I think it makes sense to focus on one cloud because you can always apply the learned principles to another providers services.

I would remove nothing from the course, but maybe go into detail about some other cloud services. For instance EFS which we had to use for our project.