Serverless Tweet Analysis
Implementing Cloud Patterns

Alaleh Hamidi

Abstract

This lab assignment provides the students with the experience of setting up a serverless data pipeline empowered by Flask framework to detect the sentiments of a given tweets and store the results according to two important cloud patterns named 'Sharding' and 'Proxy'. In this implementation, the execution time of patterns depends more on input data nature than on the pattern type. Because we didn't consider master-slave structure of Proxy pattern. So, the logic behind both patterns are the same; dividing data.

1 Introduction

Nowadays serverless computing services become more popular because they assist product engineers to innovate rapidly, in addition to alleviating the problems of system engineering and easier operational management. AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS). The Lambda functions can perform any kind of computing task, from serving web pages and processing streams of data to calling APIs and integrating with other AWS services [1]. In this lab assignment, we set up a serverless data pipeline using Amazon Lambda functions to conduct sentiment analysis of a given list of tweets. The list of tweets is accepted via a Flask RESTful web service. Flask is a micro web framework for Python which provides functionality to build web applications. In the next phases, we use AWS Lambda functions to detect the sentiment of given tweets and store the results on AWS RDS. We should implement pattern strategies to load data into AWS RDS which are named "Sharding" and "Proxy". Figure1 shows the design architecture of the sentiment analysis data pipeline in this assignment.

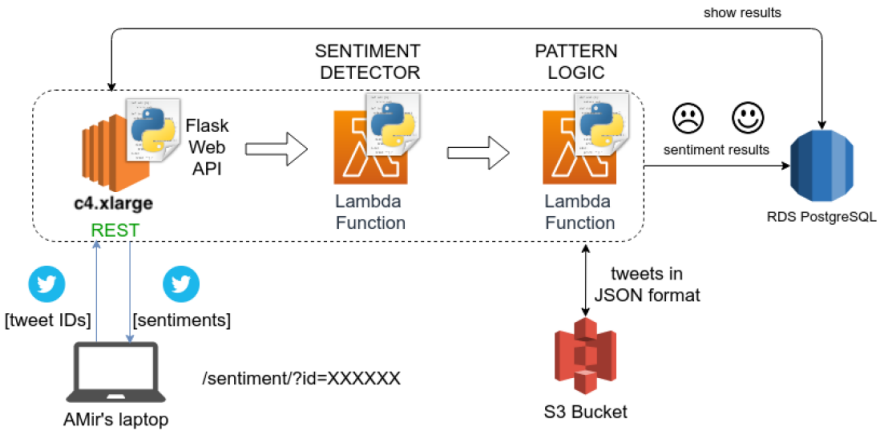

Figure 1: Design Architecture of the Sentiment Analysis Data Pipeline

2 Methodology

The main goal of this assignment is to benchmark two types of cloud patterns by obtaining the execution time of each one independently and comparing them. Since we can't implement these patterns in parallel, I found the following approaches to obtain the execution time of each pattern independently:

1. Running the application separately for each pattern by using two different Http URLs.

> *<IPv4 Public IP of EC2 instance>:5000/sharding*
> *or*
> *<IPv4 Public IP of EC2 instance>:5000/proxy*

2. Obtaining the execution time of those parts of code which are specifically related to implementing of one pattern.

> *A-applied exe-time = Total exe-time - Total B exe-time*

In the first approach, the pattern type can be sent to each phase via "Payload" parameter of "invoke" function during triggering next phase.

In the second approach, the aggregated execution-time of each pattern in each phase (Flask, Sentiment and RDS phases) can be returned to the previous phase till returning to the Flask phase to calculate the considered execution time using above formula. Only Flask and RDS phase should consider the specific execution time of each pattern, while the pattern types doesn't affect Sentiment phase execution time. Flask phase should calculate pattern-specific time during getting the result from RDS via a query. Because the time for getting the result from Proxy-related tables is different

from the Sharding-related ones. RDS phase should calculate pattern-specific time during storing the result on RDS based on the patterns.

The first approach gives more accurate execution time of the project for each independent pattern. But, it is not as user friendly as the second approach is. Since accuracy is more important for this assignment, I chose the first approach.

3 Implementation

Sentiment and RDS phases are implemented using two AWS Lambda functions respectably named "lambdahandler1" and "lambdahandler2". We run the python code written for Flask phase on an EC2 instance ($c4.xlarge$) and trigger Sentiment phase using "invoke" function which is inside the Flask phase code. The RDS phase is triggered by the invoke function inside "lambdahandler1" (related to Sentiment phase) as well.

3.1 Flask framework

To run the Flask application on the EC2 instance, we should open the port number 5000 of EC2 instance to all TCP traffic via adding a role to its security group. In addition, we should two specific packages on the EC2 instance:

$$pip3 install flask$$

$$pip3 install psycopg2-binary$$

I wrote an HTML code used as user interface while the Flask application is running. User can upload the json file as the input of application (list of tweets) via this HTML page which can be up by running the following URL:

<IPv4 Public IP of EC2 instance>:5000/<Pattern Type>

The results will be automatically downloaded as json file after the input file is uploaded by user via this HTML interface. Flask application makes the result file from the query that it sends to the RDS tables. In addition, Flask application uploads the input file (which user has uploaded) to the directory "RawFiles" of S3 instance as "input.json". Of course, it checks the file size before uploading it, because Comprehend service (which is used in the Sentiment phase) has a limitation of file size. In addition, the writing directory of lambda functions which named "tmp", is limited to 512 MB. Moreover, Flask application sends a parameter named "shard" to "lambdahandler1" which determines the type of pattern to implement. Flask application do this via "Payload" parameter of "invoke" function during triggering next phase.

3.2 Sentiment Detection

To detect the sentiment of tweets, I applied Amazon Comprehend which is a powerful natural language processing (NLP) service that uses machine learning for sentiment analysis. Using this service eliminates the needs for data cleaning and other preprocessing efforts and it performs everything by itself. Although NLTK (Natural Language Toolkit) is another powerful amazon sentiment analyzer which is faster than Amazon comprehend, but since the output of NLTK is a number between -1 and 1, and the output of Amazon Comprehend is the same output which is requested in this assignment, I preferred to choose Amazon Comprehend.

For establishing this phase, we created one lambda function named "lambdahandler1" with thr following policies:

1. ComprehendFullAccess:Provides full access to Amazon Comprehend.
2. AWSLambdaExecute: Provides Put, Get access to S3 and full access to CloudWatch Logs.
3. AWSLambdaRole: Default policy for AWS Lambda service role.

This lambda function downloads "input.json" from S3 as the input file of Amazon Comprehend. After conducting sentiment analysis using Amazon Comprehend, this lambda function uploads the results to directory "ProcessedFiles" of S3 as a json file named "Result.json". Then it triggers RDS phase and sends the "shard" parameter to "lambdahandler2" via "invoke" function.

3.3 Storing results on AWS Relational Database Service (RDS)

For this phase, I created one RDS instance and two schemas inside it. To access to RDS without any problem, RDS should be public. In addition, we should add an inbound rule (via VPC security group) to allow all traffics from anywhere have access to it. We should apply two cloud patterns named "Sharding" and "Proxy". Sharding pattern puts tweets from year 2020 in one instance, and the rest of the years into another one. Proxy pattern selects even, and odd numbers of id column representing the tweet ID and stores them in separate schema. So, I created two tables in each schema; One for Proxy and one another for Sharding (totally four tables). The lambda function delete all the content of all four tables before storing the result on them.

Moreover, I created "lambdahandler2" with the following policies to conduct this phase:

1. AmazonRDSDataFullAccess: Allows full access to use the RDS data APIs, secret store APIs for RDS database credentials, and DB console query management APIs to execute SQL statements on Aurora Serverless clusters in the AWS account.
2. AWSLambdaExecute: Provides Put, Get access to S3 and full access to CloudWatch Logs.
3. AWSLambdaRole: Default policy for AWS Lambda service role.

Since "lambdahandler2" uses psycopg2 packsge and AWS Lambda misses the required PostgreSQL libraries in the AMI image, we should install the package locally and then zip the installed package and python code to upload it on Amazon Lambda service. This function uses the "shard" parameter which Flask application and "lambdahandler1" send it respectably and at last, "lambdahandler2" use it to determine which pattern should be applied on the results to store them on RDS.

This lambda function downloads "Result.json" from S3 as the input file.

4 Results

I tried two sentiment analyzers named Amazon Comprehend and NLTK. When I applied NLTK, the execution time for Sharding and Proxy pattern were approximately the same (on average 3.9 second). But when I applied Amazon Comprehend, the average execution times for Sharding and Proxy patterns were respectably 20.52 and 19.80 second which shows not a significant difference. As we can see NLTK is significantly faster than Amazon Comprehend.

The other point is that in our implementation, it seems the execution time depends more on input dataset nature than on the pattern type and the worst case occurs when all the data should save on one table; for instance when all the data is odd (for Proxy) or occurs after 2020 (for Sharding).

I think this is because we didn't implement the exact form of Proxy and consider master-slave structure. But if we did so, I guess the Proxy execution time would be more than the Sharding. Because Proxy always has to store all the data to each table, but Sharding store less than all data on each table.

5 Tips about running this implementation

The only point user should consider is the the number of Items in input file must start from "1" as determined in the assignment (Figure2).Otherwise, the code doesn't work.

```
{
    "1":{
        "id":1246953034355298304,
        "text":"some␣sample␣tweet",
        "date": "2020-04-05␣00:12:10"
    },
    "2":{
        "id":1246490999347707905,
        "text":"this␣is␣another␣sample␣tweet",
        "date": "2020-04-06␣00:11:59"
    }
}
```

Figure 2: Sructure of input file

The other thing that I should mention is that I put the direct output of Amazon Comprehend which I saved in S3 as the "results.json" in the submition material. Because you can get the one obtained from query of Flask to RDS by running my code (It will be downloaded automatically after you upload your input file via my HTML interface).

References

[1] https://www.serverless.com/aws-lambda/