Spark Networks – Senior Data Engineer

Coding Assignment Documentation

# Introduction

The assignment is done on Ubuntu 20.x OS system, by setting up Spark environment. It can be done on a local computer/workstation or on an EC2 machine with Ubuntu AMI (with sudo privileges).

# Infra/Architecture Setup

This project is done on my personal AWS account where I have used an Ubuntu based EC2 machine and setup Apache Spark on my own rather than using AWS EMR due to cost reasons. The free tier machine t2.micro has only 1GB RAM and wasn’t enough for processing, so I have used t2.small for the assignment.

This project makes use of a Data Lake architecture where the raw and processed data is dumped to S3 rather than using an expensive service like RedShift.

I have configured 2 IAM users with limited access with the below details and the URL to access the AWS console is - <https://113911312463.signin.aws.amazon.com/console>

|  |  |  |  |
| --- | --- | --- | --- |
| IAM User | Allow | Deny | Password |
| sparknet-user-admin | Non PII buckets PII bucket Secrets Manager EC2 Read only access |  | [SparkNetworks@1](mailto:SparkNetworks@1) |
| sparknet-user-normal | Non PII buckets | PII bucket | [SparkNetworks@2](mailto:SparkNetworks@2) |

the intent being other teams who want to perform some sort of data analysis or science can access only the non-sensitive data while an elite group of trusted members can access everything.

I have prepared a config file (spark\_net.yaml) that contains all the possible config information like various S3 paths and insights and data quality checks queries.

# GitHub

The entire code base is uploaded to my personal GitHub account’s public repository named SparkNetRepo which can be found using here - <https://github.com/ShubhamMeshram/SparkNetRepo> and has 3 branches for implementation of 3 approaches explained in [Section 3](#_Data_protection_approaches)

* Branch\_appr1
* Branch\_appr2
* Branch\_appr3

# Data protection

To comply with GDPR laws to ensure data protection is given a high priority I have come up with the following 3 approaches out of which any suitable approach (with any improvisation) can be pushed into a production like setup. The general idea is to avoid keeping a few sensitive columns in plain text.

For all the 3 approaches I have chosen, ("firstName", "lastName", "address") for user dataset and “msg” for message dataset, columns to perform data protection on.

## Approach 1

This method makes use of the [cryptography](https://www.tutorialspoint.com/cryptography_with_python/cryptography_with_python_modules_of_cryptography.htm) library in python which makes use of a secret key of our choice. I have generated a 32 bytes URL-safe base64-encoded string which is a secret stored on AWS Secrets Manager with appropriate IAM access. I have created a [Spark UDF](https://sparkbyexamples.com/spark/spark-sql-udf/) which performs encryption on a given collection of columns and returns an encrypted dataframe. This can be decrypted with the same key.

## Approach 2

This method uses the [lit](https://sparkbyexamples.com/pyspark/pyspark-lit-add-literal-constant/) method in spark and just replaces the original value with a literal of our choice (\*\*\*Masked\*\*\*) in this case.

## Approach 3

This is the simplest method of all which drops the columns of our choice containing sensitive information.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Approach | Crypt Method | Branch | Common S3 Bucket | Approach specific S3 Bucket |
| Approach1 | Encrypted | branch\_appr1 | sparknet-pii-bucket | sparknet-nonpii-bucket-appr1 |
| Approach2 | Masked | branch\_appr2 | sparknet-nonpii-bucket-appr2 |
| Approach3 | Dropped | branch\_appr3 | sparknet-nonpii-bucket-appr3 |

Each branch for each approach only differs in the S3 bucket and the data protection method.

# Code Artifacts

This section contains all the relevant information related to the code base.

## Cron Scheduling

This job is scheduled for execution every night as per the below cron expressions for each branch at a difference of 5 minutes. Below screenshot displays the said information and can also be access on the EC2 machine by using the command crontab -e

A screenshot of a computer

Description automatically generated with medium confidence

sudo service cron status – ensures cron service is installed/running

## Steps to setup Spark

The steps to be followed to setup spark on an Ubuntu machine can be found here: <https://github.com/ShubhamMeshram/SparkNetRepo/blob/branch_appr1/conf/SparkSetupUbuntu.sh>

## Analytics and Insights

Various queries as per the ask are present in the config file (spark\_net.yaml) that are executed through Spark SQL and the final output is generated as a CSV which contains pandas dataframe appended side by side. Below is a screenshot of the config file:

Text

Description automatically generated

## Data Quality

The config file contains various checks to capture any anomaly or issues in the data which creates a summary output CSV file which contains DQ information on various metrics as seen in the screenshot below:

Text

Description automatically generated

The DQ checks capture NULL and duplicity issues. Different action can be taken upon seeing the DQ check summary reports like:

* Whether or not the faulty records need to be dropped
* Contact the API or external data vendor or concerned team to ensure such quality issues don’t arise at the source itself.
* Various handling logic like appending dummy ID or address or any value can be appended into the faulty column’s record.

## File structure on S3

The 3 approaches dump data to the common bucket sparknet-pii-bucket and the respective individual buckets too.

Below is an example:



Graphical user interface, application, Teams

Description automatically generated



The data is partitioned based on year month and day each time execution happens. The most recent data can always be found in the recent folder. We can travel back in time to any given day to see how the data was previously using the archive folder.

## Code Structure

Relevant scripts are present in certain folders. Below is a screenshot of how the code base looks like:

Text

Description automatically generated

Every method in every script in every folder contains [docstrings](https://www.geeksforgeeks.org/python-docstrings/) which are enough inline documentation to understand which function performs what logic and information about arguments.

The script driver.py is the important script which drives the execution and is also scheduled to run via cron. Each branch only differs in the S3 bucket and the data protection method.

## Important commands

The Public IP address will be provided by me where the git repo is already cloned. Kindly ensure to be in the directory /home/ubuntu/smeshram/SparkNetRepo

Below are a few UNIX commands to run for testing the code:

* **Clone git repo**: git clone <https://github.com/ShubhamMeshram/SparkNetRepo.git>
* **Switch Branches:** git checkout <add branch name here>
* **Run driver script:** clear && git fetch && git checkout branch\_appr1 && git pull && spark-submit code/driver.py

# Coding Best Practices

* The entire solution is built by implementing OOP and functional programming concepts so ensure the code is clean and structured well and uses functions and objects.
* Respective import statements are present various files.
* [DRY Software Development principle](https://www.digitalocean.com/community/tutorials/what-is-dry-development) – used it wherever possible to ensure code is not redundant

# Future Scope/Enhancements

Since this is an assignment and not a fully-fledged or soon-to-be production-ready project, below are a few thoughts on what can be included to make this whole solution bigger and better:

1. This can be very well done with [Apache Airflow](https://airflow.apache.org/) for orchestration and better logging by making use of various operators that airflow has to offer.
2. The current solution stores data on S3 which can have the following downstream use cases:

* Push to RedShift if a data-warehouse kinda approach is required.
* Make use of AWS Glue & a crawler to catalog datasets and then query them using AWS Athena.
* [Databricks Delta Lake](https://docs.databricks.com/delta/index.html) can be created on top of AWS S3 Data-lake which offers a plethora of services like time travel and different table types and table level locking/security etc.
* [AWS QuickSight](https://aws.amazon.com/quicksight/) - To have a complete AWS solution, BI visualization (reports/dashboard) can be done using AWS quicksight or any external tools like tableau or MicroStrategy

1. We can create trend charts using visualization libraries like [matplotlib](https://matplotlib.org/stable/users/explain/backends.html) to create a trend of various metrics to understand things like what is the pattern of users sending messages in a particular location or time frame etc.
2. To enhance the robustness of the solution, libraries like [mock](https://pypi.org/project/mock/) and [unittest](https://docs.python.org/3/library/unittest.html) can be used to create unit test and regression test scripts which need to pass each time a new feature is pushed to the branch.
3. Since this solution runs at the end of the day, daily and requires some time to execute, there could be a slight slippage of data which the API may never capture for that particular day – although it might be quite insignificant since we are not dealing with investment/banking transaction data where data of every second matters.
   * If at all we don’t want to lose data for every/any second, we can expand this solution to use the streaming service that Apache offers - Spark Streaming.