AWS POC – 1

**Git repo -**  [https://github.com/Manasa-56/AWSPoc](%20https://github.com/Manasa-56/AWSPoc)

**Task 1 (File Ingestion from Local to AWS S3 periodically):**

**Objective:**

The aim of the task is to convert all the .csv files placed in a local folder to .orc format and load to S3 Bucket.

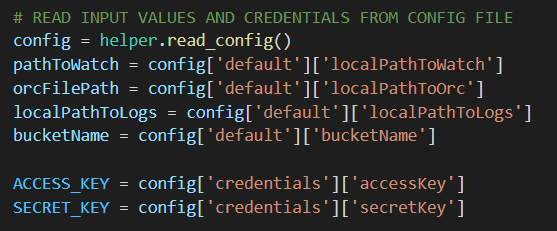
**Introduction:**

Large CSV files are received daily and placed in a local storage path. All the received files are converted to ORC format and uploaded to the specified S3 bucket. Input CSV files are generally greater than 1 GB. Few filters are applied before storing to S3.

**Tools Used: Python, AWS S3**

**Detailed Description:**

The first step is to read all the necessary inputs and credentials from the config file. The **configparser** module from Python's standard library is used to define functionality for reading and writing configuration files. This creates .ini extension file which consists for Sections and Key/Value entries.

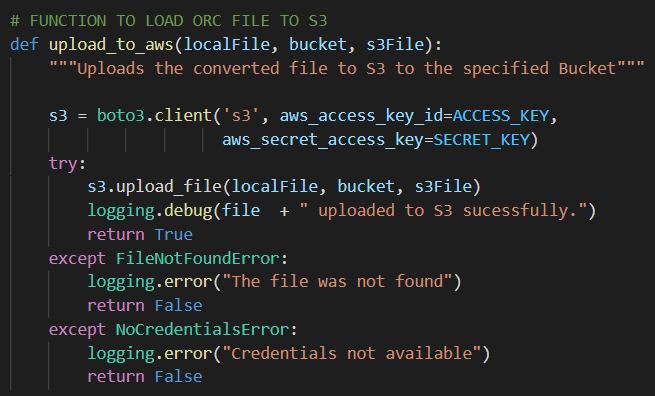


We list all the files in the specified input local folder using the l**istdir()** method. **listdir()** returns a list containing the names of the entries in the directory given by path. The list is in arbitrary order.

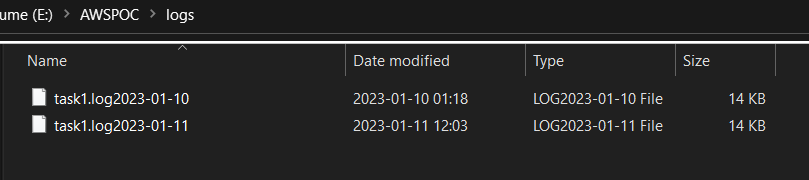


All the files present in the local directory are processed one after the other. The CSV files present in the path are converted to ORC using the **pyorc** module. It is a Python module for reading and writing [Apache ORC](https://orc.apache.org/) file format. The schema is read from the csv file and converted to orc row by row.

Once the ORC conversion is done completely, the csv file is then moved to an archive folder. Later the converted orc file is load to the defined S3 bucket.



Logging is maintained for using the logging module. This module defines functions and classes which implement a flexible event logging system for applications and libraries. Logs will be created for each run and maintained separately for each day.



This process is scheduled using the Windows Scheduler. It lets us automate tasks on Windows 10 OS. It triggers to run the program at the specified time daily. We created a .bat file containing the path where the Python exe is stored and the Path where the Python script is stored.



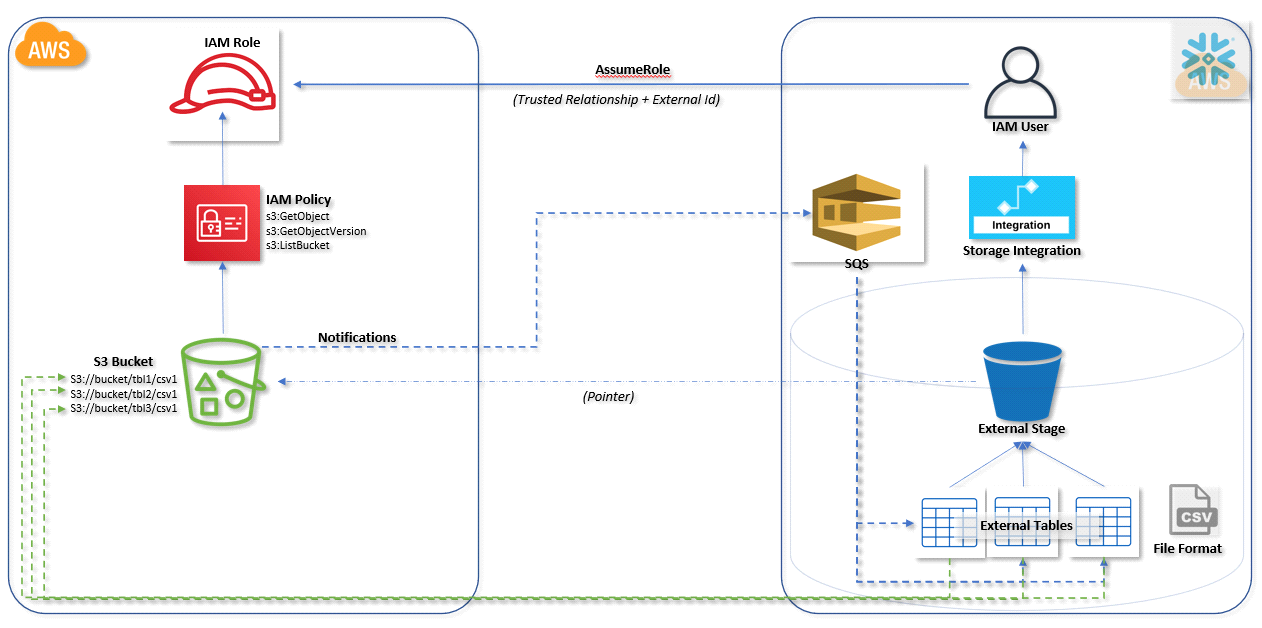
**Challenges Faced:**

Tried converting the .csv to .orc in python using the pyarrow module but faced few issues where the pyarrow.orc module is not being loaded. Alternatively used pyorc for the conversion.

**Task 2 Details:**

It is to build a triggering mechanism to automatically refresh external tables in snowflake whenever any new file loaded in Amazon s3 bucket.

**Architecture Design:**



**Process:**

1) Create IAM role for snowflake

2) Create an Amazon S3 bucket

3) Create new database in snowflake

4) Create storage integration from snowflake to the Amazon S3 bucket and add IAM role arn in STORAGE\_AWS\_ROLE\_ARN field and also give respective Amazon S3 Bucket name in STORAGE\_ALLOWED\_LOCATIONS.

query:

create or replace storage integration s3\_int

TYPE = EXTERNAL\_STAGE

STORAGE\_PROVIDER = S3

ENABLED = TRUE

STORAGE\_AWS\_ROLE\_ARN = 'arn:aws:iam::074521873501:role/Snowflake\_SNS\_Role'

STORAGE\_ALLOWED\_LOCATIONS = ('s3://snowflakeusingsns') COMMENT = 'Snowflake external table refresh'

5) Create an external stage and integrate it with Amazon S3 bucket

Query:

create stage SnowFlakeUsingSnsExternalStage

url = 's3://snowflakeusingsns'

storage\_integration = s3\_int;

6) Create file format, which is of ORC format

Query:

create or replace file format my\_orc\_format

type = orc

null\_if = ('NULL', 'null');

7) Navigate to IAM role and from there click on Trust Relationships tab and edit it by giving valid STORAGE\_AWS\_IAM\_USER\_ARN and STORAGE\_AWS\_EXTERNAL\_ID (We can fetch these details by querying the below query)  
  
DESC INTEGRATION <storage integration name>

8) Create Amazon SNS topic of standard version with proper information in Access Policy.

9) Execute the below query in snowflake and need to update SNS access policy

SELECT system$get\_aws\_sns\_iam\_policy('{<SNS ARN>}')

10 ) After Execution the above query paste the result in AWS SNS Access policy at the end.

11) Create Event Notification for the S3 bucket, So for that navigate to S3 bucket -> Properties->Event Notifications -> Create it by giving proper Event types and Destination i.e SNS topic and specify SNS topic, that created above.

12) Create External table in snowflake by specifying sns topic

Query:

create or replace external table s3\_to\_snowflake\_using\_sns.public.test\_dataset

with location = @SnowFlakeUsingSnsExternalStage file\_format = 'my\_orc\_format'

13) Once after the external table created, snowflake internal SQS will be generated and we can see that in by navigating to Amazon SNS that created above.

14) upload a file in Amazon S3 bucket and that will trigger SNS with the Event that created in S3. Now this SNS will broadcast the event notification to snowflake internal sqs and sqs will sync the uploaded data to external table.

**Technologies used:**

1) Snowflake

2)AWS Services

**Challenges Faced:**

While creating Event notification, initially we could not able to specify the SNS topic, that is because the s3 bucket and SNS topic were in different regions. But later, we made sure that both are in same locations and that solved our problem.

**Task 3 (Data Transformation):**

**Objective:**

The aim of the task is to transform the data in first RDS and load it in the second RDS.

**Introduction:**

The data in the MySQL database is fetched and basic transformations are performed and loaded into the PostgreSQl database using the EMR.

**Chart, diagram

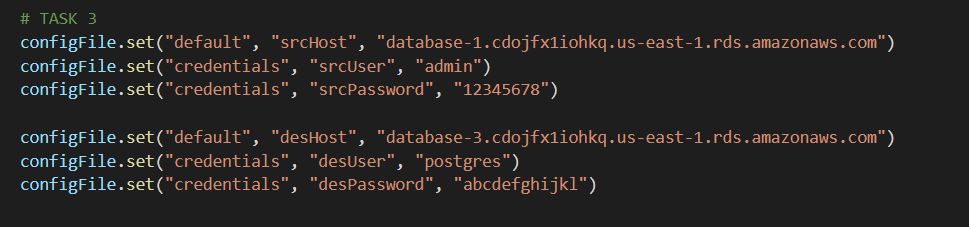
Description automatically generated**

**Architecture**

**Tools used: Python, AWS RDS, EMR**

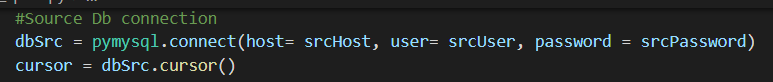
**Detailed Description:**

Step 1: Created two RDS instances one for the source (MySQL) and the other for the destination (PostgreSQL). The host, credentials are noted and passed to the configuration file.



Step 2: The first step is to read all the necessary inputs and credentials from the config file. The credentials and required parameters for the connection to databases are read from the config file.

Step 3: Imported pymysql module for establishing the connection with the MySQL database. Once the connection is made successfully a database ‘RdsDataBase’ and ‘employeeData´ table are created. Sample data is inserted into the table ‘employeeData´ table.

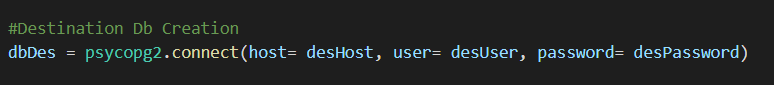


Step 4: After the data is inserted successfully, the data is queried using the Pandas dataframe. Basic transformations like filtering are applied.

Text

Description automatically generated

Step 5: A new connection is then made to the target PostgreSQl database using the ‘psycopg2’ module. Then a database ‘PostgresDestDataBase’ is created followed by a ‘employeeDestData’ table to store the data.



Step 6: The transformed is then loaded to the destination table ‘employeeDestData’.

**Challenges Faced:**

Tried connecting to On-prime SQL Server but observed few connectivity issues. So implemented with the RDS MySQL database as a source.