Guided Capstone Step Five

Step Five: Pipeline Orchestration

Now that you have all of your functionality implemented, you need to orchestrate your pipeline, connecting all of the individual components into an end-to-end data pipeline. In the real world, data processing jobs are typically organized into automated workflows.

There are many ways to define a workflow and it's scope, but typically workflows are repeatable pipeline units which can be run independently. In cloud architecture, a workflow can be run in an elastic Hadoop cluster with input and output data that are persistent on cloud storage such as Azure Blob Storage. Your guided pipeline can be divided into two workflows:

- Preprocessing Workflow: data ingestion and batch load
- Analytical Workflow: analytical ETL job

In this project, you'll launch your Spark job using a command line shell script in each workflow.

Learning Objectives:

By the end of this step, you'll:

- Write shell script to submit Spark jobs
- Execute Spark jobs in Azure Elastic Clusters
- Design job status tracker

Prerequisites:

- Azure Elastic Cluster
- Knowledge of command line

5.1 Preprocessing Workflow

Use a shell script to call a data ingestion Spark job:

```
#!/bin/sh
. ~/.bash_profile
spark-submit \
--master local \
--py-files dist/guidedcapstone-1.0-py3.7.egg \
--jars jars/postgresql-42.2.14.jar, jars/hadoop-azure.jar, jars/azure-storage.jar
 \
etl/src/run_data_ingestion.py \
config/config.ini
```

5.2 Analytical Workflow

Use shell script to call an analytical ETL job:

```
#!/bin/sh
. ~/.bash_profile
spark-submit \
--master local \
--py-files dist/guidedcapstone-1.0-py3.7.egg \
--jars jars/postgresql-42.2.14.jar, jars/hadoop-azure.jar, jars/azure-storage.jar \
etl/src/run_reporter.py \
config/config.ini
```

5.3 Job Status Tracking

In the daily operation of a product, failures in system or application level are expected to occur from time to time. Maintaining job status is very important in order to keep track of successful and failed job runs, so you can then take proper actions.

Typical job status is maintained in a database table which supports transactional operations like insert, update, delete, etc.

5.3.1 Defining A Job Status Table

An entry to your job status table should uniquely identify the actual job which runs on a certain date, since all of your jobs run only once per day.

Your job status table should contain these fields:

- Job_id (primary and unique): naming convention can be jobname_yyyy-mm-dd
- Status: success or failure state of the job
- Updated_time: datetime of the entry

5.3.2 Defining A Job Management Class

This Python utility class will consist of three major functionalities:

- Assign_job_id
- Update_job_status
- Get_job_status.

```
import datetime
import psycopg2

class Tracker(object):
    """
    job_id, status, updated_time
    """
    def __init__(self, jobname, dbconfig):
        self.jobname = jobname
        self.dbconfig = dbconfig

    def assign_job_id(self):
        # [Construct the job ID and assign to the return variable]
        return job_id

    def update_job_status(self, status):

        job_id = self.assign_job_id()
        print("Job ID Assigned: {}".format(job_id))
        update_time = datetime.datetime.now()
        table_name = self.dbconfig.get("postgres", "job_tracker_table_name")
        connection = self.get_db_connection()

        try:
            # [Execute the SQL statement to insert to job status table]
        except (Exception, psycopg2.Error) as error:
            print("error executing db statement for job tracker.")
        return

    def get_job_status(self, job_id):
        # connect db and send sql query
        table_name = self.dbconfig.get('postgres', 'job_tracker_table_name')
        connection = self.get_db_connection()
        try:
```

```
            record = # [Execute SQL query to get the record]
            return record
        except (Exception, psycopg2.Error) as error:
            print("error executing db statement for job tracker.")
        return

    def get_db_connection(self):
        connection = None
        try:
            connection = # [Initialize database connection]
        except (Exception, psycopg2.Error) as error:
            print("Error while connecting to PostgreSQL", error)

        return connection
```

5.3.3 Populate Job Status Table During The Job Run

The status that is populated is based on whether the job ran successfully without error. The call of reporter.report method in this example is the Spark job that delivers the result. It populates "success" after the call, assuming it completes without error. If an error was caught, the status "failed" is populated instead.

```
def run_reporter_etl(my_config):
    trade_date = my_config.get('production', 'processing_date')
    reporter = Reporter(spark, my_config)

    tracker = Tracker('analytical_etl', my_config)
    try:
        reporter.report(spark, trade_date, eod_dir)
        tracker.update_job_status("success")
    except Exception as e:
        print(e)
        tracker.update_job_status("failed")
    return
```

Summary

In this step, you practised a basic end-to-end operation on the pipeline you built. Shell scripts are usually the most straightforward, reliable way to trigger a job. You implemented job status tracking, which gives you the ability to monitor the progress of the daily job run, in order to make sure that results are always delivered.

Open Questions:

- Can you tweak the job status tracker implementation to prevent parallel run (multiple instances of the same job running simultaneously)?
- Can you tweak the job status tracker implementation to prevent an analytical ETL job from running before the data ingestion has finished?

Submit this assignment:

- Commit and push the updated code to Github and submit to your mentor to review.