UDACITY

< Return to "Data Engineering Nanodegree" in the classroom

Data Lake

REVIEW

CODE REVIEW 4

HISTORY

Meets Specifications

Brilliant Udacity Learner,

Congratulations! 🎉
You've successfully passed all the specification in **one shot**! I must admit that the structure of this project implementation is impressive! You should be proud of the work done as you seem to have a good hold on **Spark** and **Data Lakes** to build an ETL pipeline for a data lake hosted on **S3**.

It was my pleasure reviewing this wonderful project. Please continue with this same spirit of hard work in the projects ahead.

Extra Materials

Below are some additional links to help you deepen your understanding on the related concepts:

- Best Practices for Building a Data Lake in Amazon S3 and Amazon Glacier
- Data Lake Implementation: Processing and Querying Data in Place
- Data Lakes and Analytics on AWS
- Building Big Data Storage Solutions (Data Lakes) for Maximum Flexibility
- The Business And Technological Benefits Of Data Lakes
- Build a data lake on Google Cloud Platform (GCP)

ETL

The script, etl.py, runs in the terminal without errors. The script reads song_data and load_data from S3, transforms them to create five different tables, and writes them to partitioned parquet files in table directories on S3.

Well done! The etl.py script runs successfully without errors. It reads the data from S3, transforms it to the fact and dimension tables and writes it to partitioned parquet files on S3 as required.👍

Each of the five tables are written to parquet files in a separate analytics directory on S3. Each table has its own folder within the directory. Songs table files are partitioned by year and then artist. Time table files are partitioned by year and month. Songplays table files are partitioned by year and month.

Awesome! All the parquet files are correctly setup to save in the s3 bucket. 👏

The **partitionBy** argument in `write.parquet()` was correctly utilized to implement the following:

- Songs table files are partitioned by year and then artist.
- Time table files are partitioned by year and month.
- Songplays table files are partitioned by year and month.

Suggestions

- What is the difference between partitionBy and repartition methods in Spark?
- How to read and write Parquet files in Spark

- A nice discussion about Writing data to Parquet with partitions
- stackoverflow: How to save a partitioned parquet file in Spark 2.1

Each table includes the right columns and data types. Duplicates are addressed where appropriate.

Nice work. Duplicates were addressed appropriately by using dropDuplicates() method. Keep it up!

Suggestions

- Duplicated rows in dataframes could be also avoided upon reading data files before creating your parquet files:
  For example:
  - song data file:
    `df = spark.read.json(song_data).dropDuplicates()`
  - log data file:
    `df = spark.read.json(log_data).dropDuplicates()`
  - song data to use for songplays table: `song_df = spark.read.json(input_data + "song-data/*/*/*/TRA*.json").dropDuplicates()`

Extra Materials

- Deduplicating and Collapsing Records in Spark DataFrames
- Drop duplicates by some condition
- Stackoverflow: Spark SQL DataFrame - distinct() vs dropDuplicates()

Code Quality

The README file includes a summary of the project, how to run the Python scripts, and an explanation of the files in the repository. Comments are used effectively and each function has a docstring.

Impressive README file! It contains all the necessary details for a writeup. I would also suggest using docstrings to effectively explain each functions. Though there are no standard and rules for doing so, please note that it is an essential part that documenting your code is going to serve well enough for writing clean code and well written programs.

writing clean code and well-written programs.

Suggestions

To make your writeup even better, you may include screenshots of your final tables, and add more details about your project, such as the dataset and how did you clean the data. Any in-line comments that were clearly part of the project instructions should be removed, and parameter descriptions and data types can be also included in your docstrings.

For example:

```
def process_song_data(spark, input_data, output_data):
    """
    Read song data and process it and save to provided output location
    :param spark: Spark session
    :param input_data: Input url
    :param output_data: Output location
    """
```

You may check some links below to know more about python code documentation:

- Documenting Python Code: A Complete Guide
- PEP 257 -- Docstring Conventions
- Python Docstrings
- Different types of writing Docstrings

Scripts have an intuitive, easy-to-follow structure with code separated into logical functions. Naming for variables and functions follows the PEP8 style guidelines.

Most of the code is well optimized with intuitive and easy-to-follow structure which follows PEP8 style guidelines but please limit all lines to a maximum of 79 characters. The Python standard library is conservative and requires limiting lines to 79 characters (and docstrings/comments to 72). You may use backslash "\" for line continuation.

Suggestions

- You may find this link helpful to Check your code for PEP8 requirements.
- Here are some additional resource to know more about PEP8 style guidelines:
    - PEP 8: Style Guide for Python Code
    - How to Write Beautiful Python Code With PEP 8
    - PEP-8 Tutorial: Code Standards in Python

DOWNLOAD PROJECT

4 CODE REVIEW COMMENTS ›

RETURN TO PATH

Rate this review