Springboard

Data Engineering Career Track
Guided Capstone Step Two: Data Ingestion

Data Ingestion Instructions

Introduction

After you have created the data structure diagram, you can start the build process. For a data processing system, the first step is to ingest the data sources. Your Spring Capital data sources come from stock exchange daily submissions files in a semi-structured text format. This means the records follow a certain formatting convention like JSON, but don't obey a tabular structure formatted for a relational database. The data ingestion process parses the semi-structured data out so it can be loaded into Spark.

Learning Objectives:

- Learn to parse CSV and JSON files
- Create a Spark DataFrame with defined schema
- Persist the Spark DataFrame into file system using partitioning

Prerequisites:

- Python: basics, string manipulation, control flow, exception handling, JSON parsing
- PySpark: RDD from text file, custom DataFrames, write with partitions, Parquet

Azure Jars:

- Download and store Azure jar files (hadoop-azure.jar and azure-storage.jar) in the **jars** folder.

2.1 Data Source

The data that's submitted by exchanges will be in two different formats: CSV and JSON. CSV means Comma Separated Values, so it is a text document that contains many values separated by commas. JSON is JavaScript Object Notation, in which text data is stored following a standard convention. Both of these are flat text files containing trade and quotes from different numbers of fields. The record type can be identified by column 'rec_type' which is a fixed position for CSV files.

Source files will be stored in cloud storage (Azure Blob Storage) using the below locations:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master('local').appName('app').getOrCreate()
spark.conf.set(
"fs.azure.account.key.<storage-account-name>.blob.core.windows.net",
      "<your-storage-account-access-key>"
)
raw =
spark.textFile("wasbs://<container-name>@<storage-account-name>.blob.core.w
indows.net/<path_in_container>")
parsed = raw.map(lambda line: parse_json(line))
data = spark.createDataFrame(parsed)
```

2.2 Parsing The Source Data

In this step, you'll write a PySpark parser to parse the out values from a flat text file. The parser will read the text file line by line and transform it into either trade or quote records based on "rec_type" column value. The parser will also handle corrupted records, allowing the program to continue processing the good records without crashing. These bad records will be put aside into a special partition.

2.2.1 Define The CSV Parser

Since CSV records have a common schema, parsing CSV files is straightforward. Simply use the string-split method to get the list of columns and identify the record type. Any exceptions that occur during parsing are automatically labeled as bad records.

```
from typing import List
def parse_csv(line:str):
   record_type_pos = 2
   record = line.split(",")
   try:
```

```
    # [logic to parse records]
    if record[record_type_pos] == "T":
        event = common_event(col1_val, col2_val, ..., "T","")
        return event
    elif record[record_type_pos] == "Q":
        event = common_event(col1_val, col2_val, … , "Q","")
        return event
  except Exception as e:
    # [save record to dummy event in bad partition]
    # [fill in the fields as None or empty string]
    return common_event(,,,,....,,,,,"B",line)
```

2.2.2 Use The CSV Parser In Spark Transformation To Process The Source Data

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master('local').appName('app').getOrCreate()
spark.conf.set(
"fs.azure.account.key.<storage-account-name>.blob.core.windows.net",
      "<your-storage-account-access-key>"
)
raw =
spark.textFile("wasbs://<container-name>@<storage-account-name>.blob.core.w
indows.net/<path_in_container>")
parsed = raw.map(lambda line: parse_csv(line))
data = spark.createDataFrame(parsed)
```

2.2.3 Define The JSON Parser

Unlike CSV, JSON records are organized by key-value pairs. The parser should retrieve the value of the columns using their column names as keys. Python provides an easy to use JSON utility.

```
import json
def parse_json(line:str):
  record_type = record['event_type']
  try:
    # [logic to parse records]
    if record_type == "T":
      # [Get the applicable field values from json]
      if # [some key fields empty]
        event = common_event(col1_val, col2_val, ..., "T","")
      else :
```

```
            event = ommon_event(„„....„„„"B",line)
        return event
    elif record_type == "Q":
        # [Get the applicable field values from json]
        if # [some key fields empty]
            event = common_event(col1_val, col2_val, ... , "Q","")
        else :
            event = ommon_event(„„....„„„"B",line)
        return event
  except Exception as e:
    # [save record to dummy event in bad partition]
    # [fill in the fields as None or empty string]
    return common_event(„„....„„„"B",line)
```

2.2.4 Use The JSON Parser In Spark Transformation To Process The Source Data

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master('local').appName('app').getOrCreate()
spark.conf.set(
"fs.azure.account.key.<storage-account-name>.blob.core.windows.net",
      "<your-storage-account-access-key>"
)
raw =
spark.textFile("wasbs://<container-name>@<storage-account-name>.blob.core.w
indows.net/<path_in_container>")
parsed = raw.map(lambda line: parse_json(line))
data = spark.createDataFrame(parsed)
```

2.3 Populate Temporary Trade And Quote Data

2.3.1 Define A Common Event Schema

Since you have two different records types, you'll need to define a common record schema to hold them both. This common record schema must have a field to partition trade and quote data later on in the process.

Common Event

Column	Type
trade_dt	DateType
rec_type	StringType

symbol	StringType
exchange	StringType
event_tm	TimestampType
event_seq_nb	IntegerType
arrival_tm	TimestampType
trade_pr	DecimalType
bid_pr	DecimalType
bid_size	IntegerType
ask_pr	DecimalType
ask_size	IntegerType
partition	StringType

2.3.2 Write The Common Events Into Partitions As Parquet Files To HDFS

It is typical to create a method to Write with partitioning, which is used to separate one dataset into multiple parts in one pass. Here you'll use the column "partition" as a partition key.

```
data.write.partitionBy("partition").mode("overwrite").parquet("output_dir")
```

At this point, we'll see daily processed data stored in this data structure:

```
output_dir/partition=T/
output_dir/partition=Q/
output_dir/partition=B/
```

2.4 Summary

Now that you've completed the Data Ingestion steps, here are the main takeaways to keep in mind for future implementations. If you feel any of the takeaways are still confusing to you, please discuss them with your mentor in the next weekly call.

- Use of a custom record parser is necessary when reading variant schema files or unstructured files.
- Write as partition is a one pass method to split one dataset into multiple. In Big Data problems, using a one pass method if possible is always prefered.

In addition, here are some open questions for you to reflect on:

- What are other ways to split the dataset into multiple parts other than partitioning?
- What would be the difference in performance?

Submit this assignment:

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