Springboard


# Data Engineering Career Track
## Guided Capstone Step Four

### Step Four: Analytical ETL

Now that you have loaded the trade and quote tables with daily records, this step focuses on using SparkSQL and Python to build an ETL job that calculates the following results for a given day:

- Latest trade price before the quote (i.e. Yesterday's close price in case of first quote and for every other quote within the day, you are expected to retrieve last price when the trade was executed for the given stock).
- Latest 30-minute moving average trade price, before the quote (this means 30-minutes before the quote. So in the case of the first quote of the day, you will look at the last 30 minutes of the previous day.).
- The bid/ask price movement from the previous day's closing price.

**Learning Objectives:**

By the end of this step, you will be able to:

- Use more advanced Spark data transformations and SQL functions.
- Leverage different methods to join datasets.
- Use Spark Broadcast to achieve optimal performance.
- Use Spark UI to monitor Spark jobs.

**Prerequisites:**

- PySpark: temporary view, creating a DataFrame from a Hive table, knowledge of broadcast variables, use of SparkUI
- SQL: analytical functions, joins, unions
- Hive: Hive DDL, external tables, etc.

### 4.1 Read Parquet Files From Azure Blob Storage Partition

Use Spark to read Parquet files from the corresponding Azure blob partition. For example, if our current date is “2020-07-29”, we should find the location with that date on Azure blob.

```
df =
spark.read.parquet("cloud-storage-path/trade/date={}".format("2020-07-29"))
```

### 4.2 Create Trade Staging Table

As described, we are providing results for each quote record but using trade records as reference. In this scenario, it’s easier to first create a staging table with all the results calculated based on the trade dataset alone. We can name the table “tmp_trade_moving_avg”.

#### 4.2.1 Use Spark To Read The Trade Table With Date Partition “2020-07-29”

```
df = spark.sql("select symbol, event_tm, event_seq_nb, trade_pr from trades
where trade_dt = '2020-07-29'")
```

#### 4.2.2 Create A Spark Temporary View

```
df.createOrReplaceTempView("tmp_trade_moving_avg")
```

#### 4.2.3 Calculate The 30-min Moving Average Using The Spark Temp View

- Use an analytical or window function if you are using Spark SQL. Partition by symbol and order by time. The window should contain all records within 30-min of the corresponding row.

```
mov_avg_df = spark.sql("""
    select symbol, exchange, event_tm, event_seq_nb, trade_pr,
    # [logic to derive last 30 min moving average price] as mov_avg_pr
    from tmp_trade_moving_avg
    """)
```

#### 4.2.4 Save The Temporary View Into Hive Table For Staging

```
mov_avg_df.write.saveAsTable("temp_trade_moving_avg")
```

### 4.3 Create Staging Table For The Prior Day’s Last Trade

You’ll need another staging table in order to make the last trade price available to the quote records of corresponding symbols. Note that this table has only 1 price per trade/exchange, so the record size will be small enough.

#### 4.3.1 Get The Previous Date Value

```
import datetime
date = datetime.datetime.strptime('2020-08-04', '%Y-%m-%d')
prev_date_str = [use datetime utility to calculate previous date]
```

#### 4.3.2 Use Spark To Read The Trade Table With Date Partition "2020-07-28"

```
df = spark.sql("select symbol, event_tm, event_seq_nb, trade_pr from trades
where trade_dt = '{}'".format(prev_date_str))
```

#### 4.3.3 Create Spark Temporary View

```
df.createOrReplaceTempView("tmp_last_trade")
```

#### 4.3.4 Calculate Last Trade Price Using The Spark Temp View

```
last_pr_df = spark.sql("""select symbol, exchange, last_pr from (select
symbol, exchange, event_tm, event_seq_nb, trade_pr,
# [logic to derive last 30 min moving average price] AS last_pr
FROM tmp_trade_moving_avg) a
""")
```

#### 4.3.4 Save The Temporary View Into Hive Table For Staging

```
mov_avg_df.write.saveAsTable("temp_last_trade")
```

### 4.4 Populate The Latest Trade and Latest Moving Average Trade Price To The Quote Records

Now that you've produced both staging tables, join them with the main table "quotes" to populate trade related information.

#### 4.4.1 Join With Table temp_trade_moving_avg

You need to join "quotes" and "temp_trade_moving_avg" to populate trade_pr and mov_avg_pr into quotes. However, you cannot use equality join in this case; trade events don't happen at the same quote time. You want the latest in time sequence. This is a typical time sequence analytical use case. A good method for this problem is to merge both tables in a common time sequence.

#### 4.4.1.1 Define A Common Schema Holding "quotes" and "temp_trade_moving_avg" Records

This is a necessary step before the union of two datasets which have a different schema (denormalization). The schema needs to include all the fields of quotes and temp_trade_mov_avg so that no information gets lost.

New schema fields:

| Column | Value |
|---|---|
| trade_dt | Value from corresponding records |
| rec_type | “Q” for quotes, “T” for trades |
| symbol | Value from corresponding records |
| event_tm | Value from corresponding records |
| event_seq_nb | From quotes, null for trades |
| exchange | Value from corresponding records |
| bid_pr | From quotes, null for trades |
| bid_size | From quotes, null for trades |
| ask_pr | From quotes, null for trades |
| ask_size | From quotes, null for trades |
| trade_pr | From trades, null for quotes |
| mov_avg_pr | From trades, null for quotes |

#### 4.4.1.2 Create Spark Temp View To Union Both Tables

Perform data normalization in the select statement from both table followed by union:

```
quote_union = spark.sql("""
   # [Creates a denormalized view union both quotes and
temp_trade_moving_avg, populate null for fields not applicable]
   """)
quote_union.createOrReplaceTempView("quote_union")
```

#### 4.4.1.3 Populate The Latest trade_pr and mov_avg_pr

Use a window analytical function to populate the values from the latest records with rec_type “T.” The window should be under the partition of the exchange symbol:

```
quote_union_update = spark.sql("""
   select *,
   # [logic for the last not null trade price] AS last_trade_pr,
   # [logic for the last not null mov_avg_pr price] AS last_mov_avg_pr
   from quote_union
   """)
quote_union_update.createOrReplaceTempView("quote_union_update")
```

#### 4.4.1.4 Filter For Quote Records

Since you ultimately only need quote records, filter out trade events after the calculation.

```
quote_update = spark.sql("""
   select trade_dt, symbol, event_tm, event_seq_nb, exchange,
   bid_pr, bid_size, ask_pr, ask_size, last_trade_pr, last_mov_avg_pr
   from quote_union_update
   where rec_type = 'Q'
   """)
quote_update.createOrReplaceTempView("quote_update")
```

#### 4.4.2 Join With Table temp_last_trade To Get The Prior Day Close Price

The prior day close price table has a single record per symbol and exchange. For this join, you can use equality join since the join fields are only “symbol” and “exchange”. However, note that this table has a very limited number of records (no more than the number of symbol and exchange combinations). This is an excellent opportunity to use broadcast join to achieve optimal join performance. Broadcast join is always recommended if one of the join tables is small enough so that the whole dataset fits in memory.

```
quote_final = spark.sql("""
   select trade_dt, symbol, event_tm, event_seq_nb, exchange,
   bid_pr, bid_size, ask_pr, ask_size, last_trade_pr, last_mov_avg_pr,
   bid_pr - close_pr as bid_pr_mv, ask_pr - close_pr as ask_pr_mv
   from (
   # [Broadcast temp_last_trade table. Use quote_update to left outer join
temp_last_trade]
   ) a
   """)
```

#### 4.4.3 Write The Final Dataframe Into Azure Blob Storage At Corresponding Partition

The final output needs to be stored as a new partition for the object on Azure Blob Storage.

```
quote_update.write.parquet("cloud-storage-path/quote-trade-analytical/date=
{}".format(trade_date))
```

### 4.5 Summary

This step is the core of the data pipeline, as it implements all the analytical ETL logic. It demonstrates your logical thinking by breaking down the complex problem into simpler steps, which is key for engineering tasks. Given the complexity of the processing logic, there may be performance issues that require tuning. For Big Data applications, this is an unavoidable problem. Here are the main takeaways:

- When possible, use temporary tables to maintain simple and clean code.
- Analytical functions are powerful tools for data processing logic, especially time sequence problems.
- Union and filter could be an option for inequality join use cases.
- Use broadcast join to avoid full data shuffle when any dataset is small enough.

**Open questions**

- Can you use a regular SQL join for 3.4.1?
- If you don't specify the "BROADCAST" hint in the join query, what would happen? How can you tell whether the broadcast join is in effect?
- If our data submission doesn't happen on weekends and holidays, how would you get the previous date?

**Submit This Assignment:**

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