databricks Silver Table Creation (current version)

(https://databricks.com)

Set the currently used database to the shared group database

```
databaseName = "group4_finalproject"
print('databaseName: ' + databaseName)

bronzeTableCreationNotebook = "Bronze Table Creation"

if not spark.catalog.databaseExists(databaseName):
  print(f"Database {databaseName} does not exists. Please run {bronzeTableCreationNotebook} first to create the database a
else:
  spark.sql(f"use {databaseName}")
```

```
databaseName: group4_finalproject
```

Train Data

Load the train_bronze delta table into a dataframe

```
train_bronze_df = spark.table("train_bronze")
```

```
display(train_bronze_df)
train_bronze_df.printSchema()
dbutils.data.summarize(train_bronze_df)
print(f"Train Dataframe Count: {train_bronze_df.count()}")
```

Show that the first day of data in the dataframe (2021-09-01) corresponds to data_block_id 0

```
display(train_bronze_df.filter("data_block_id = 0"))
```

There are 528 null values in the target column which we drop and create an intermediate dataframe, train_silver_df

```
train_bronze_df_nulls = train_bronze_df.filter("target IS NULL")
display(train_bronze_df_nulls)
```

```
train_silver_df = train_bronze_df.dropna(how='any')
```

```
print(f"Train Dataframe Count: {train_silver_df.count()}")
```

```
Train Dataframe Count: 2017824
```

County 12, which is listed as "unknown" in the county mapping, makes up around 30k of 2 million+ data points, and so we drop it. Without knowing the county, we won't be able to add weather information to these rows.

```
train_silver_df.filter("county = 12").count()
```

```
train_silver_df = train_silver_df.filter(train_silver_df["county"] != 12)
```

```
train_silver_df.createOrReplaceTempView("temp_table")
merge_column = "row_id"

non_merge_columns = [col for col in train_silver_df.columns if col != f"{merge_column}"]
update_conditions = " OR ".join([f"destination.{col} <> source.{col}" for col in non_merge_columns])
update_set = ", ".join([f"destination.{col} = source.{col}" for col in non_merge_columns])

merge_sql = f"""
MERGE INTO train_silver AS destination
     USING temp_table AS source
        ON destination.{merge_column} = source.{merge_column}

      WHEN MATCHED AND ({update_conditions}) THEN
           UPDATE SET {update_set}
      WHEN NOT MATCHED BY TARGET THEN
           INSERT *
      WHEN NOT MATCHED BY SOURCE THEN
           DELETE
"""


if not spark._jsparkSession.catalog().tableExists("train_silver"):
    train_silver_df.write.format("delta").option("mergeSchema",
"true").partitionBy("data_block_id").saveAsTable("train_silver")
else:
    spark.sql(merge_sql)
```

Client Data

Load the client_bronze delta table into a dataframe

```
client_bronze_df = spark.table("client_bronze")
```

Display the dataframe, along with its schema, summary statistics, and count

```
display(client_bronze_df)
client_bronze_df.printSchema()
dbutils.data.summarize(client_bronze_df)
print(f"Client Dataframe Count: {client_bronze_df.count()}")
```

Show that the first day of data in the dataframe (2021-09-01) corresponds to data_block_id 2

```
display(client_bronze_df.filter("data_block_id = 2"))
```

Rename columns to aid in joining and rename dataframe client_silver_df

```
client_bronze_df.columns
```

```
client_silver_df = client_bronze_df.withColumnRenamed("date", "client_date").withColumnRenamed("product_type",
"client_product_type").withColumnRenamed("county", "client_county").withColumnRenamed("is_business",
"client_is_business").withColumnRenamed("data_block_id", "client_data_block_id")
```

```
client_silver_df.createOrReplaceTempView("temp_table")
merge_columns = ["client_product_type", "client_county", "client_is_business", "client_date"]

non_merge_columns = [col for col in client_silver_df.columns if col not in merge_columns]

update_conditions = " OR ".join([f"destination.{col} <> source.{col}" for col in non_merge_columns])
update_set = ", ".join([f"destination.{col} = source.{col}" for col in non_merge_columns])

on_clause = " AND ".join([f"destination.{col} = source.{col}" for col in merge_columns])

merge_sql = f"""
MERGE INTO client_silver AS destination
     USING temp_table AS source
        ON {on_clause}

      WHEN MATCHED AND ({update_conditions}) THEN
           UPDATE SET {update_set}
      WHEN NOT MATCHED BY TARGET THEN
           INSERT *
      WHEN NOT MATCHED BY SOURCE THEN
           DELETE
"""

if not spark._jsparkSession.catalog().tableExists("client_silver"):
    client_silver_df.write.format("delta").option("mergeSchema",
"true").partitionBy("client_data_block_id").saveAsTable("client_silver")
else:
    spark.sql(merge_sql)
```

Electricity Prices Data

Load the electricity_prices_bronze delta table into a dataframe

```
electricity_prices_bronze_df = spark.table("electricity_prices_bronze")
```

Display the dataframe, along with its schema, summary statistics, and count

```
display(electricity_prices_bronze_df)
electricity_prices_bronze_df.printSchema()
dbutils.data.summarize(electricity_prices_bronze_df)
print(f"Dataframe Count: {electricity_prices_bronze_df.count()}")
```

The first day of data isn't available until data_block_id 1, meaning the data is delayed by 1 day

```
display(electricity_prices_bronze_df.filter("data_block_id = 1"))
```

Change name of "forecast_date" column to "electricity_effective_datetime", since "forecast_date" is misleading as the prices are not forecasts but rather are just effective at a future date.

```
electricity_prices_bronze_df = electricity_prices_bronze_df.withColumnRenamed("forecast_date",
"electricity_effective_datetime")
```

Import pyspark functions used in code below

```
from pyspark.sql.functions import col, expr
```

Create electricity_prices_silver_df with new column, "electricity_available_datetime" with a value equal to 1 day after "electricity_effective_datetime", which represents the hour of the day that each row is for but on the day the data is available, for aid in matching with wide_silver_df.

```
electricity_prices_silver_df = electricity_prices_bronze_df.withColumn("electricity_available_datetime",
expr("electricity_effective_datetime + INTERVAL 1 DAY"))
```

Rename columns that will be joined into wide table to aid with identifying them

```
electricity_prices_silver_df.columns
```

```
['electricity_effective_datetime',
 'euros_per_mwh',
 'origin_date',
 'data_block_id',
 'electricity_available_datetime']
```

```
electricity_prices_silver_df = electricity_prices_silver_df.withColumnRenamed("euros_per_mwh",
"electricity_euros_per_mwh").withColumnRenamed("origin_date",
"electricity_origin_date").withColumnRenamed("data_block_id", "electricity_data_block_id")
```

```
electricity_prices_silver_df.createOrReplaceTempView("temp_table")
merge_column = "electricity_effective_datetime"

non_merge_columns = [col for col in electricity_prices_silver_df.columns if col != f"{merge_column}"]
update_conditions = " OR ".join([f"destination.{col} <> source.{col}" for col in non_merge_columns])
update_set = ", ".join([f"destination.{col} = source.{col}" for col in non_merge_columns])

merge_sql = f"""
MERGE INTO electricity_prices_silver AS destination
     USING temp_table AS source
        ON destination.{merge_column} = source.{merge_column}

      WHEN MATCHED AND ({update_conditions}) THEN
           UPDATE SET {update_set}
      WHEN NOT MATCHED BY TARGET THEN
           INSERT *
      WHEN NOT MATCHED BY SOURCE THEN
           DELETE
"""

if not spark._jsparkSession.catalog().tableExists("electricity_prices_silver"):
    electricity_prices_silver_df.write.format("delta").option("mergeSchema",
"true").partitionBy("electricity_data_block_id").saveAsTable("electricity_prices_silver")
else:
    spark.sql(merge_sql)
```

Gas Prices Data

Load the gas_prices_bronze delta table into a dataframe

```
gas_prices_bronze_df = spark.table("gas_prices_bronze")
```

Display the dataframe, along with its schema, summary statistics, and count

```
display(gas_prices_bronze_df)
gas_prices_bronze_df.printSchema()
dbutils.data.summarize(gas_prices_bronze_df)
print(f"Dataframe Count: {gas_prices_bronze_df.count()}")
```

Rename forecast_date column to the more accurate "gas_effective_date"

```
gas_prices_bronze_df = gas_prices_bronze_df.withColumnRenamed("forecast_date", "gas_effective_date")
```

The first day of data isn't available until data_block_id 1, meaning the data is delayed by 1 day

```
display(gas_prices_bronze_df.filter("data_block_id = 1"))
```

```
display(gas_prices_bronze_df.filter("data_block_id = 2"))
```

Change column names to make them easier to identify after joining gas_prices with wide table

```
gas_prices_silver_df = gas_prices_bronze_df.withColumnRenamed("lowest_price_per_mwh",
"gas_lowest_price_per_mwh").withColumnRenamed("highest_price_per_mwh",
"gas_highest_price_per_mwh").withColumnRenamed("origin_date", "gas_origin_date").withColumnRenamed("data_block_id",
"gas_data_block_id")
```

```
gas_prices_silver_df.columns
```

```
['gas_effective_date',
 'gas_lowest_price_per_mwh',
 'gas_highest_price_per_mwh',
 'gas_origin_date',
 'gas_data_block_id']
```

```
gas_prices_silver_df.createOrReplaceTempView("temp_table")
merge_column = "gas_effective_date"

non_merge_columns = [col for col in gas_prices_silver_df.columns if col != f"{merge_column}"]
update_conditions = " OR ".join([f"destination.{col} <> source.{col}" for col in non_merge_columns])
update_set = ", ".join([f"destination.{col} = source.{col}" for col in non_merge_columns])

merge_sql = f"""
MERGE INTO gas_prices_silver AS destination
     USING temp_table AS source
        ON destination.{merge_column} = source.{merge_column}

      WHEN MATCHED AND ({update_conditions}) THEN
           UPDATE SET {update_set}
      WHEN NOT MATCHED BY TARGET THEN
           INSERT *
      WHEN NOT MATCHED BY SOURCE THEN
           DELETE
"""

if not spark._jsparkSession.catalog().tableExists("gas_prices_silver"):
    gas_prices_silver_df.write.format("delta").option("mergeSchema",
"true").partitionBy("gas_data_block_id").saveAsTable("gas_prices_silver")
else:
    spark.sql(merge_sql)
```

Weather Station to County Mapping Data

Load the weather_station_to_county_bronze delta table into a dataframe

```
weather_station_to_county_mapping_bronze_df = spark.table("weather_station_to_county_mapping_bronze")
```

Display the dataframe, along with its schema, summary statistics, and count

```
display(weather_station_to_county_mapping_bronze_df )
weather_station_to_county_mapping_bronze_df.printSchema()
dbutils.data.summarize(weather_station_to_county_mapping_bronze_df)
```

Use geopy to determine closest county to the weather stations with no county assigned and assign them to that county

```
!pip install geopy
```

```
Note: you may need to restart the kernel using dbutils.library.restartPython() to use updated packages.
Collecting geopy
  Downloading geopy-2.4.1-py3-none-any.whl (125 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 125.4/125.4 kB 2.8 MB/s eta 0:00:00
Collecting geographiclib<3,>=1.52
  Downloading geographiclib-2.0-py3-none-any.whl (40 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 40.3/40.3 kB 4.9 MB/s eta 0:00:00
Installing collected packages: geographiclib, geopy
Successfully installed geographiclib-2.0 geopy-2.4.1
Note: you may need to restart the kernel using dbutils.library.restartPython() to use updated packages.
```

```
from geopy.distance import geodesic
```

```
location_county_no_nulls = weather_station_to_county_mapping_bronze_df.selectExpr("Round(latitude, 1) AS latitude",
"longitude", "county", "county_name").dropna(how='any')
display(location_county_no_nulls)
```

Table

	latitude	longitude	county	county_name
1	58.2	22.2	10	Saaremaa
2	58.5	22.2	10	Saaremaa
3	58.5	22.7	10	Saaremaa
4	58.8	22.7	1	Hiiumaa
5	58.5	23.2	10	Saaremaa
6	58.5	23.7	7	Pärnumaa
7	58.8	23.7	6	Läänemaa

49 rows

```
labeled_locations = location_county_no_nulls.rdd.map(tuple).collect()
labeled_locations
```

```
[(58.2, 22.2, 10, 'Saaremaa'),
 (58.5, 22.2, 10, 'Saaremaa'),
 (58.5, 22.7, 10, 'Saaremaa'),
 (58.8, 22.7, 1, 'Hiiumaa'),
 (58.5, 23.2, 10, 'Saaremaa'),
 (58.5, 23.7, 7, 'Pärnumaa'),
 (58.8, 23.7, 6, 'Läänemaa'),
 (59.1, 23.7, 6, 'Läänemaa'),
 (58.5, 24.2, 7, 'Pärnumaa'),
 (58.8, 24.2, 9, 'Raplamaa'),
 (59.1, 24.2, 0, 'Harjumaa'),
 (59.4, 24.2, 0, 'Harjumaa'),
 (58.2, 24.7, 7, 'Pärnumaa'),
 (58.5, 24.7, 7, 'Pärnumaa'),
 (58.8, 24.7, 9, 'Raplamaa'),
```

```
(59.1, 24.7, 9, 'Raplamaa'),
(59.4, 24.7, 0, 'Harjumaa'),
(58.2, 25.2, 7, 'Pärnumaa'),
(58.5, 25.2, 14, 'Viljandimaa'),
(58.8, 25.2, 3, 'Järvamaa'),
```

```
location_county_nulls = weather_station_to_county_mapping_bronze_df.filter("county IS NULL")
location_county_nulls = location_county_nulls.selectExpr("ROUND(latitude, 1) AS latitude", "longitude")
display(location_county_nulls)
```

Table

	latitude	longitude
1	57.6	21.7
2	57.9	21.7
3	58.2	21.7
4	58.5	21.7
5	58.8	21.7
6	59.1	21.7
7	59.4	21.7

63 rows

```
unlabeled_locations = location_county_nulls.rdd.map(tuple).collect()
unlabeled_locations
```

```
[(57.6, 21.7),
 (57.9, 21.7),
 (58.2, 21.7),
 (58.5, 21.7),
 (58.8, 21.7),
 (59.1, 21.7),
 (59.4, 21.7),
 (59.7, 21.7),
 (57.6, 22.2),
 (57.9, 22.2),
 (58.8, 22.2),
 (59.1, 22.2),
 (59.4, 22.2),
 (59.7, 22.2),
 (57.6, 22.7),
 (57.9, 22.7),
 (58.2, 22.7),
 (59.1, 22.7),
 (59.4, 22.7),
 (59.7, 22.7),
 (57.6, 23.2),
```

```
def find_closest_label(location, labeled_locations):
    closest_label = None
    min_distance = float('inf')

    for labeled_location in labeled_locations:
        distance = geodesic(location, labeled_location[:2]).kilometers
        if distance < min_distance:
            min_distance = distance
            closest_county = labeled_location[2]
            closest_county_name = labeled_location[3]

    return closest_county, closest_county_name
```

```
assigned_counties = []

for location in unlabeled_locations:
    county, county_name = find_closest_label(location, labeled_locations)
    assigned_counties.append((county_name, location[1], location[0], county))
    print(f"Location {location} is closest to {county_name}, county number {county}")
```

```
Location (57.6, 21.7) is closest to Saaremaa, county number 10
Location (57.9, 21.7) is closest to Saaremaa, county number 10
Location (58.2, 21.7) is closest to Saaremaa, county number 10
Location (58.5, 21.7) is closest to Saaremaa, county number 10
Location (58.8, 21.7) is closest to Saaremaa, county number 10
Location (59.1, 21.7) is closest to Hiiumaa, county number 1
Location (59.4, 21.7) is closest to Hiiumaa, county number 1
Location (59.7, 21.7) is closest to Hiiumaa, county number 1
Location (57.6, 22.2) is closest to Saaremaa, county number 10
Location (57.9, 22.2) is closest to Saaremaa, county number 10
Location (58.8, 22.2) is closest to Hiiumaa, county number 1
Location (59.1, 22.2) is closest to Hiiumaa, county number 1
Location (59.4, 22.2) is closest to Hiiumaa, county number 1
Location (59.7, 22.2) is closest to Hiiumaa, county number 1
Location (57.6, 22.7) is closest to Saaremaa, county number 10
Location (57.9, 22.7) is closest to Saaremaa, county number 10
Location (58.2, 22.7) is closest to Saaremaa, county number 10
Location (59.1, 22.7) is closest to Hiiumaa, county number 1
Location (59.4, 22.7) is closest to Läänemaa, county number 6
Location (59.7, 22.7) is closest to Läänemaa, county number 6
Location (57.6, 23.2) is closest to Saaremaa, county number 10
```

```
from pyspark.sql import Row
rows = [Row(county_name=entry[0], longitude=entry[1], latitude=entry[2], county=entry[3]) for entry in
assigned_counties]

assigned_counties_df = spark.createDataFrame(rows)
```

```
display(assigned_counties_df)
```

Table

	county_name	longitude	latitude	county
1	Saaremaa	21.7	57.6	10
2	Saaremaa	21.7	57.9	10
3	Saaremaa	21.7	58.2	10
4	Saaremaa	21.7	58.5	10
5	Saaremaa	21.7	58.8	10
6	Hiiumaa	21.7	59.1	1
7	Hiiumaa	21.7	59.4	1

63 rows

```
weather_station_to_county_mapping_silver_df =
weather_station_to_county_mapping_bronze_df.dropna(how='any').union(assigned_counties_df)
```

```
display(weather_station_to_county_mapping_silver_df)
```

Table

	county_name	longitude	latitude	county

1	Saaremaa	22.2	58.2	10
2	Saaremaa	22.2	58.49999999999999	10
3	Saaremaa	22.7	58.49999999999999	10
4	Hiiumaa	22.7	58.79999999999999	1
5	Saaremaa	23.2	58.49999999999999	10
6	Pärnumaa	23.7	58.49999999999999	7
7	Läänemaa	23.7	58.79999999999999	6

112 rows

```
weather_station_to_county_mapping_silver_df.printSchema()
```

```
weather_station_to_county_mapping_silver_df = weather_station_to_county_mapping_silver_df.selectExpr("county_name",
"longitude", "ROUND(latitude, 1) AS latitude", "county")
```

```
weather_station_to_county_mapping_silver_df.display()
```

Save weather_station_to_county_mapping_silver_df to a silver table

```
weather_station_to_county_mapping_silver_df.write.format("delta").mode("overwrite").saveAsTable("weather_station_to_
county_mapping_silver")
```

Historical Weather Data

Load the historical_weather_bronze delta table into a dataframe

```
historical_weather_bronze_df = spark.table("historical_weather_bronze")
```

Display the dataframe, along with its schema, summary statistics, and count

```
display(historical_weather_bronze_df)
historical_weather_bronze_df.printSchema()
dbutils.data.summarize(historical_weather_bronze_df)
```

Load weather_station_to_county_mapping_silver_df from table and join with historical_weather on latitude, longitude to include county info with the historical weather data

```
weather_station_to_county_mapping_silver_df = spark.table("weather_station_to_county_mapping_silver")
```

```
weather_station_to_county_mapping_silver_df.display()
```

```
historical_weather_silver_df = historical_weather_bronze_df.join(weather_station_to_county_mapping_silver_df,
["latitude", "longitude"], "left")
```

```
display(historical_weather_silver_df)
```

Show that the historical weather data for each day is split across 2 data_block_ids, meaning the 1st 11 hours are available on one day and the remaining hours are available on the next day

```
display(historical_weather_silver_df.orderBy('datetime', 'data_block_id').filter("data_block_id = 1"))
```

Table

	latitude	longitude	datetime	temperature	dewpoint	rain	snowfall	surface_pres
1	57.6	21.7	2021-09-01T00:00:00Z	14.4	12	0	0	1015.8
2	57.6	22.2	2021-09-01T00:00:00Z	14	12	0	0	1010.6
3	57.6	22.7	2021-09-01T00:00:00Z	14.4	12.8	0	0	1014.9
4	57.6	23.2	2021-09-01T00:00:00Z	15.4	13	0	0	1014.4
5	57.6	23.7	2021-09-01T00:00:00Z	15.9	12.6	0	0	1013.8
6	57.6	24.2	2021-09-01T00:00:00Z	13.1	10.6	0	0	1013.4
7	57.6	24.7	2021-09-01T00:00:00Z	13	10.7	0	0	1005

1,232 rows

```
display(historical_weather_silver_df.orderBy('datetime', 'data_block_id').filter("data_block_id = 2"))
```

Assign data from the 1st 11 hours and 2nd 13 hours to different availability days to be able to join with wide table

```
from pyspark.sql.functions import col, expr

historical_weather_silver_df = historical_weather_silver_df.withColumn(
    "historical_weather_available_datetime",
    expr("CASE WHEN HOUR(datetime) < 11 THEN datetime + INTERVAL 1 DAY ELSE datetime + INTERVAL 2 DAY END")
)
```

```
historical_weather_silver_df.columns
```

```
historical_weather_silver_df.display()
```

```
historical_weather_silver_df.count()
```

Check for null values

```
if (historical_weather_silver_df.dropna().count() == historical_weather_silver_df.count()) == True:
  print("No nulls")
else:
  print("Nulls present, need to investigate")
```

```
No nulls
```

```
historical_weather_silver_df.columns
```

```
['latitude',
 'longitude',
 'datetime',
 'temperature',
 'dewpoint',
 'rain',
 'snowfall',
 'surface_pressure',
 'cloudcover_total',
 'cloudcover_low',
 'cloudcover_mid',
 'cloudcover_high',
 'windspeed_10m',
 'winddirection_10m',
 'shortwave_radiation',
 'direct_solar_radiation',
 'diffuse_radiation',
 'data_block_id',
 'county_name',
 'county',
 'historical_weather_available_datetime']
```

```
historical_weather_silver_df.createOrReplaceTempView("temp_table")
merge_columns = ["latitude", "longitude", "historical_weather_available_datetime", "datetime"]

non_merge_columns = [col for col in historical_weather_silver_df.columns if col not in merge_columns]

update_conditions = " OR ".join([f"destination.{col} <> source.{col}" for col in non_merge_columns])
update_set = ", ".join([f"destination.{col} = source.{col}" for col in non_merge_columns])

on_clause = " AND ".join([f"destination.{col} = source.{col}" for col in merge_columns])

merge_sql = f"""
MERGE INTO historical_weather_silver AS destination
     USING temp_table AS source
        ON {on_clause}

      WHEN MATCHED AND ({update_conditions}) THEN
           UPDATE SET {update_set}
      WHEN NOT MATCHED BY TARGET THEN
           INSERT *
      WHEN NOT MATCHED BY SOURCE THEN
           DELETE
"""

if not spark._jsparkSession.catalog().tableExists("historical_weather_silver"):
    historical_weather_silver_df.write.format("delta").option("mergeSchema",
"true").partitionBy("data_block_id").saveAsTable("historical_weather_silver")
else:
    spark.sql(merge_sql)
```

Forecast Weather Data

Load the forecast_weather_bronze delta table into a dataframe

```
forecast_weather_bronze_df = spark.table("forecast_weather_bronze")
```

Display the dataframe, along with its schema, summary statistics, and count

```
display(forecast_weather_bronze_df)
forecast_weather_bronze_df.printSchema()
dbutils.data.summarize(forecast_weather_bronze_df)
```

Check for null values

```
for column in forecast_weather_bronze_df.columns:
  print(f"Column {column} has {forecast_weather_bronze_df.filter(f'{column} is NULL').count()} nulls")
```

Drop null values

```
print("before dropping nulls, the number of data points:",  forecast_weather_bronze_df.count())
forecast_weather_silver_df = forecast_weather_bronze_df.dropna()
print("after dropping nulls, the number of data points:",  forecast_weather_silver_df.count())
```

```
before dropping nulls, the number of data points: 3424512
after dropping nulls, the number of data points: 3424510
```

Assign county names, county ids using latitude/longitude of each forecast with mapping determined by updated weather station to county mapping

```
forecast_weather_silver_df = forecast_weather_silver_df.join(weather_station_to_county_mapping_silver_df,
["latitude", "longitude"], "left")
```

```
display(forecast_weather_silver_df)
```