demo8_workbook_MASTER

October 15, 2022

1 MASTER wk8 Demo - Advanced Spark - DataFrames and Spark SQL

MIDS w261: Machine Learning at Scale | UC Berkeley School of Information | Fall 2022

So far we've been using Spark's low level APIs. In particular, we've been using the RDD (Resilient Distiributed Datasets) API to implement Machine Learning algorithms from scratch. This week we're going to take a look at how Spark is used in a production setting. We'll look at DataFrames, SQL, and UDFs (User Defined Functions). As discussed previously, we still need to understand the internals of Spark and MapReduce in general to write efficient and scalable code.

In class today we'll get some practice working with larger data sets in Spark. We'll start with an introduction to efficiently storing data and approach a large dataset for analysis. After that we'll discuss a ranking problem which was covered in Chapter 6 of the High Performance Spark book and how we can apply that to our problem. We'll follow up with a discussion on things that could be done to make this more effiicent. * … **describe** differences between data serialization formats. * … **choose** a data serialization format based on use case. * … **describe** DataFrames API, GroupBy and *Spark SQL*. * … **describe** and **create** a data pipeline for analysis. * … **use** a user defined function (UDF). * … **understand** feature engineering and aggregations in Spark.

Additional Resources: Writing performant code in Spark requires a lot of thought. Holden's High Performance Spark book covers this topic very well. In addition, Spark - The Definitive Guide, by Bill Chambers and Matei Zaharia, provides some recent developments.

[1]:
```
## Imports
import re
import json
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

from pyspark.sql import SparkSession

app_name = "week8_demo"
master = "local[*]"
spark = SparkSession\
        .builder\
```

```
        .appName(app_name)\
        .master(master)\
        .config("spark.ui.port","42229")\
        .getOrCreate()
sc = spark.sparkContext

## Change the working directory
!cd /media/notebooks/student-workspace/LiveSessionMaterials/wk08Demo_DataFrames
```

```
:: loading settings :: url = jar:file:/usr/lib/spark/jars/ivy-2.4.0.jar!/org/apa
che/ivy/core/settings/ivysettings.xml
```

```
Ivy Default Cache set to: /root/.ivy2/cache
The jars for the packages stored in: /root/.ivy2/jars
graphframes#graphframes added as a dependency
org.apache.spark#spark-avro_2.12 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent-2825d7bc-
faf1-4b84-8258-67801c3c6190;1.0
        confs: [default]
        found graphframes#graphframes;0.8.2-spark3.1-s_2.12 in spark-packages
        found org.slf4j#slf4j-api;1.7.16 in central
        found org.apache.spark#spark-avro_2.12;3.1.3 in central
        found org.spark-project.spark#unused;1.0.0 in central
:: resolution report :: resolve 325ms :: artifacts dl 7ms
        :: modules in use:
        graphframes#graphframes;0.8.2-spark3.1-s_2.12 from spark-packages in
[default]
        org.apache.spark#spark-avro_2.12;3.1.3 from central in [default]
        org.slf4j#slf4j-api;1.7.16 from central in [default]
        org.spark-project.spark#unused;1.0.0 from central in [default]
        ---------------------------------------------------------------------
        |                  |            modules            ||   artifacts   |
        |       conf       | number| search|dwnlded|evicted|| number|dwnlded|
        ---------------------------------------------------------------------
        |      default     |   4   |   0   |   0   |   0   ||   4   |   0   |
        ---------------------------------------------------------------------
:: retrieving :: org.apache.spark#spark-submit-parent-2825d7bc-
faf1-4b84-8258-67801c3c6190
        confs: [default]
        0 artifacts copied, 4 already retrieved (0kB/8ms)
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
22/10/08 15:30:25 INFO org.apache.spark.SparkEnv: Registering MapOutputTracker
22/10/08 15:30:25 INFO org.apache.spark.SparkEnv: Registering BlockManagerMaster
22/10/08 15:30:25 INFO org.apache.spark.SparkEnv: Registering
BlockManagerMasterHeartbeat
22/10/08 15:30:25 INFO org.apache.spark.SparkEnv: Registering
```

```
OutputCommitCoordinator
```

```
/bin/sh: 1: cd: can't cd to /media/notebooks/student-
workspace/LiveSessionMaterials/wk08Demo_DataFrames
```

[3]:
```python
## Load the data
data = spark.read.format('bigquery') \
  .option('table', 'bigquery-public-data:samples.gsod') \
  .load()
```

1.1 DataFrames API

Let's showcase some of the important methods that we have available when working with DataFrames

[3]:
```python
## show
data.show()
```

```
22/06/20 15:32:50 WARN org.apache.spark.sql.catalyst.util.package: Truncated the
string representation of a plan since it was too large. This behavior can be
adjusted by setting 'spark.sql.debug.maxToStringFields'.
[Stage 0:>                                                          (0 + 1) / 1]
```

```
+--------------+-----------+----+-----+---+------------------+-----------------
----+------------------+--------------------------+----------------------+------
----------------------------+---------------------+----------------------------
----+------------------+---------------------------+-----------------+---------
------------------+-------------------------+--------------------+--------------
--+-------------------------+---------------+-------------------------+-----------
--------+----------+-----+-----+-----+-----+-------+-------+
|station_number|wban_number|year|month|day|
mean_temp|num_mean_temp_samples|     mean_dew_point|num_mean_dew_point_samples|me
an_sealevel_pressure|num_mean_sealevel_pressure_samples|mean_station_pressure|nu
m_mean_station_pressure_samples|    mean_visibility|num_mean_visibility_samples|
mean_wind_speed|num_mean_wind_speed_samples|max_sustained_wind_speed|max_gust_wi
nd_speed|    max_temperature|max_temperature_explicit|min_temperature|min_tempera
ture_explicit|total_precipitation|snow_depth|  fog| rain| snow|
hail|thunder|tornado|
+--------------+-----------+----+-----+---+------------------+-----------------
----+------------------+--------------------------+----------------------+------
----------------------------+---------------------+----------------------------
----+------------------+---------------------------+-----------------+---------
------------------+-------------------------+--------------------+--------------
--+-------------------------+---------------+-------------------------+-----------
--------+----------+-----+-----+-----+-----+-------+-------+
|         39730|      99999|1929|   10| 20|  52.79999923706055|
4|              45.5|                         4|                   null|
null|                 null|                                 null| 6.199999809265137|
4|21.200000762939453|                          4|       29.899999618530273|
```

```
null|             50.0|                   false|           null|
null|               0.0|      null|false|false|false|false|  false|  false|
|         33110|      99999|1929|   12| 18|              47.5|
4|            44.0|                        4|                   null|
null|                null|                            null|              2.5|
4|            11.0|                        4|                      13.0|
null|             45.0|                   false|           null|
null|              null|      null|false|false|false|false|  false|  false|
|         37770|      99999|1931|    4| 24|  50.20000076293945|
4| 44.29999923706055|                        4|                   null|
null|                null|                            null| 5.900000095367432|
4|12.300000190734863|                        4|      18.100000381469727|
null|             45.0|                   false|           null|
null|              null|      null|false|false|false|false|  false|  false|
|        726810|      24131|1931|    6| 23|   65.0999984741211|
24|             41.5|                        8|                   null|
null|                null|                            null| 48.29999923706055|
24| 7.199999809265137|                        24|      11.100000381469727|
null|53.400001525878906|                   true|           null|
null|               0.0|      null|false|false|false|false|  false|  false|
|        726810|      24131|1931|    3|  2|  42.79999923706055|
24|             31.5|                        8|                   null|
null|                null|                            null|  72.9000015258789|
24| 2.299999952316284|                        24|       4.099999904632568|
null|32.400001525878906|                   true|           null|
null|               0.0|      null|false|false|false|false|  false|  false|
|        726810|      24131|1931|    9| 17|              67.0|
24|             40.5|                        8|                   null|
null|                null|                            null| 33.79999923706055|
24|2.4000000953674316|                        24|                     6.0|
null| 51.29999923706055|                   true|           null|
null|               0.0|      null|false|false|false|false|  false|  false|
|        726810|      24131|1931|    8|  7|   68.4000015258789|
24| 37.20000076293945|                        8|                   null|
null|                null|                            null|27.899999618530273|
24|              3.5|                        24|                     7.0|
null| 52.29999923706055|                   true|           null|
null|               0.0|      null|false|false|false|false|  false|  false|
|        726810|      24131|1932|    7| 14|   64.0999984741211|
24|54.099998474121094|                        8|                   null|
null|                null|                            null|             41.0|
24| 4.199999809265137|                        24|       8.899999618530273|
null|55.400001525878906|                   true|           null|
null|              null|      null|false|false|false|false|  false|  false|
|        726810|      24131|1932|   10| 23| 41.099998474121094|
24|             31.0|                        8|                   null|
null|                null|                            null|             41.0|
24| 4.300000190734863|                        24|      15.899999618530273|
```

```
null|35.400001525878906|                    true|           null|
null|              null|      null|false|false|false|false|  false|  false|
|        726810|      24131|1932|    1|  5| 24.600000381469727|
24|21.100000381469727|                      8|                 null|
null|                null|                            null|14.800000190734863|
24| 3.200000047683716|                      24|       4.099999904632568|
null|21.399999618530273|                    true|           null|
null|              null|      null| true| true| true| true|   true|   true|
|        726815|      24106|1932|    8| 27|              71.0|
24|              null|                      null|                 null|
null|                null|                            null|28.600000381469727|
24|               8.0|                      24|                    15.0|
null|62.400001525878906|                    true|           null|
null|               0.0|      null|false|false|false|false|  false|  false|
|        726810|      24131|1932|    8| 20|              71.0|
24| 41.70000076293945|                      8|                 null|
null|                null|                            null|              32.5|
24| 4.099999904632568|                      24|       9.899999618530273|
null|53.400001525878906|                    true|           null|
null|               0.0|      null|false|false|false|false|  false|  false|
|        726810|      24131|1932|    5| 21|  55.20000076293945|
24|46.599998474121094|                      8|                 null|
null|                null|                            null|22.600000381469727|
24| 5.300000190734863|                      24|                     8.0|
null|46.400001525878906|                    true|           null|
null|              null|      null|false|false|false|false|  false|  false|
|        370310|      99999|1933|   10| 17|  55.29999923706055|
4|              null|                      null|                 null|
null|                null|                            null|               3.0|
4|16.799999237060547|                      4|      18.100000381469727|
null|              45.0|                    false|          null|
null|               0.0|      null|false|false|false|false|  false|  false|
|        292310|      99999|1933|   12|  1|             -11.0|
4|              null|                      null|                 null|
null|                null|                            null|               4.0|
4|               5.0|                      4|       8.899999618530273|
null|             -27.0|                    true|           null|
null|              null|      null|false|false|false|false|  false|  false|
|        370310|      99999|1933|    6| 17|  62.70000076293945|
4|              null|                      null|                 null|
null|                null|                            null|              null|
null|               1.0|                      4|       1.899999976158142|
null|              55.0|                    true|           null|
null|               0.0|      null|false|false|false|false|  false|  false|
|        239330|      99999|1933|    6|  4|              70.5|
4|              null|                      null|                 null|
null|                null|                            null| 1.899999976158142|
4|               5.0|                      4|       8.899999618530273|
```

```
null|              59.0|                   false|            null|
null|                0.0|      null| true| true| true| true|   true|   true|
|        282750|      99999|1933|    1|  7|               -5.0|
4|              null|                    null|                 null|
null|                 null|                            null|3.4000000953674316|
4| 3.200000047683716|                             4|       8.899999618530273|
null|             -22.0|                    true|            null|
null|                0.0|      null|false|false|false|false|  false|  false|
|        292310|      99999|1933|    3| 17|-10.300000190734863|
4|              null|                    null|                 null|
null|                 null|                            null| 1.100000023841858|
4|              13.5|                             4|      18.100000381469727|
null|             -36.0|                    true|            null|
null|                0.0|      null|false|false|false|false|  false|  false|
|        370310|      99999|1933|    4| 23|               65.0|
4|              null|                    null|                 null|
null|                 null|                            null|              null|
null|               4.5|                          4|       8.899999618530273|
null|              52.0|                   false|            null|
null|                0.0|      null|false|false|false|false|  false|  false|
+--------------+-----------+----+-----+---+-------------------+-----------------
----+------------------+---------------------------+----------------------+------
----------------------------+----------------------+----------------------------
----+------------------+---------------------------+-------------------+---------
------------------+-------------------------+--------------------+----------------
--+-------------------------+----------------+----------------------+-----------
--------+----------+-----+-----+-----+-----+-------+-------+
only showing top 20 rows
```

Here we see `.show()`, a method that works similarly to Pandas `.head()`. You can observe that the DataFrame is stored in text, that way it's easier to distribute throughout the different executors. If you want to better display the results, we can transform the output using `.limit(n)` to a Pandas Dataframe

[4]:
```python
data.limit(10).toPandas().head()
```

[4]:
```
   station_number  wban_number  year  month  day  mean_temp  \
0           39730        99999  1929     10   20  52.799999
1           33110        99999  1929     12   18  47.500000
2           37770        99999  1931      4   24  50.200001
3          726810        24131  1931      6   23  65.099998
4          726810        24131  1931      3    2  42.799999

   num_mean_temp_samples  mean_dew_point  num_mean_dew_point_samples  \
0                      4       45.500000                           4
1                      4       44.000000                           4
```

```
2                    4      44.299999                          4
3                   24      41.500000                          8
4                   24      31.500000                          8

   mean_sealevel_pressure  …  min_temperature  min_temperature_explicit  \
0                     NaN  …              NaN                      None
1                     NaN  …              NaN                      None
2                     NaN  …              NaN                      None
3                     NaN  …              NaN                      None
4                     NaN  …              NaN                      None

   total_precipitation  snow_depth    fog   rain   snow   hail  thunder  \
0                  0.0         NaN  False  False  False  False    False
1                  NaN         NaN  False  False  False  False    False
2                  NaN         NaN  False  False  False  False    False
3                  0.0         NaN  False  False  False  False    False
4                  0.0         NaN  False  False  False  False    False

   tornado
0    False
1    False
2    False
3    False
4    False

[5 rows x 31 columns]
```

This is a public dataset from NOAA, regarding weather stations across the United States. It has a total of 31 columns.

Another important command is `.printSchema()` to check columns names and what type of data is stored on it

[5]:
```
data.printSchema()
```

```
root
 |-- station_number: long (nullable = false)
 |-- wban_number: long (nullable = true)
 |-- year: long (nullable = false)
 |-- month: long (nullable = false)
 |-- day: long (nullable = false)
 |-- mean_temp: double (nullable = true)
 |-- num_mean_temp_samples: long (nullable = true)
 |-- mean_dew_point: double (nullable = true)
 |-- num_mean_dew_point_samples: long (nullable = true)
 |-- mean_sealevel_pressure: double (nullable = true)
 |-- num_mean_sealevel_pressure_samples: long (nullable = true)
 |-- mean_station_pressure: double (nullable = true)
```

```
 |-- num_mean_station_pressure_samples: long (nullable = true)
 |-- mean_visibility: double (nullable = true)
 |-- num_mean_visibility_samples: long (nullable = true)
 |-- mean_wind_speed: double (nullable = true)
 |-- num_mean_wind_speed_samples: long (nullable = true)
 |-- max_sustained_wind_speed: double (nullable = true)
 |-- max_gust_wind_speed: double (nullable = true)
 |-- max_temperature: double (nullable = true)
 |-- max_temperature_explicit: boolean (nullable = true)
 |-- min_temperature: double (nullable = true)
 |-- min_temperature_explicit: boolean (nullable = true)
 |-- total_precipitation: double (nullable = true)
 |-- snow_depth: double (nullable = true)
 |-- fog: boolean (nullable = true)
 |-- rain: boolean (nullable = true)
 |-- snow: boolean (nullable = true)
 |-- hail: boolean (nullable = true)
 |-- thunder: boolean (nullable = true)
 |-- tornado: boolean (nullable = true)
```

[7]:

```
%%time
## To look how many data points, we can use the command .count()
print(f"Number of rows is {data.count()} and number of columns is {len(data.
 ↪columns)}")
```

```
[Stage 5:>                                                          (0 + 4) / 4]

Number of rows is 114420316 and number of columns is 31
CPU times: user 2.71 ms, sys: 2.83 ms, total: 5.54 ms
Wall time: 1.53 s
```

114 million rows! Try to fit that into a Pandas DataFrame!. Now let's check how can we filter our dataframe and how can we create new columns.

We need to lever a very important set of Spark built-in functions from `pyspark.sql.functions`, typically called `F` functions

[27]:

```
# Using built-in Spark functions are always more efficient
from pyspark.sql import types
import pyspark.sql.functions as F

## Let's create a new column called time
data = data.withColumn("time",
                  F.concat(F.col("year"),
                  F.lit("-"), F.col("month"),
                  F.lit("-"), F.col("day")) \
                  .cast(types.TimestampType()))
```

```
data.printSchema()
```

```
root
 |-- station_number: long (nullable = false)
 |-- wban_number: long (nullable = true)
 |-- year: long (nullable = false)
 |-- month: long (nullable = false)
 |-- day: long (nullable = false)
 |-- mean_temp: double (nullable = true)
 |-- num_mean_temp_samples: long (nullable = true)
 |-- mean_dew_point: double (nullable = true)
 |-- num_mean_dew_point_samples: long (nullable = true)
 |-- mean_sealevel_pressure: double (nullable = true)
 |-- num_mean_sealevel_pressure_samples: long (nullable = true)
 |-- mean_station_pressure: double (nullable = true)
 |-- num_mean_station_pressure_samples: long (nullable = true)
 |-- mean_visibility: double (nullable = true)
 |-- num_mean_visibility_samples: long (nullable = true)
 |-- mean_wind_speed: double (nullable = true)
 |-- num_mean_wind_speed_samples: long (nullable = true)
 |-- max_sustained_wind_speed: double (nullable = true)
 |-- max_gust_wind_speed: double (nullable = true)
 |-- max_temperature: double (nullable = true)
 |-- max_temperature_explicit: boolean (nullable = true)
 |-- min_temperature: double (nullable = true)
 |-- min_temperature_explicit: boolean (nullable = true)
 |-- total_precipitation: double (nullable = true)
 |-- snow_depth: double (nullable = true)
 |-- fog: boolean (nullable = true)
 |-- rain: boolean (nullable = true)
 |-- snow: boolean (nullable = true)
 |-- hail: boolean (nullable = true)
 |-- thunder: boolean (nullable = true)
 |-- tornado: boolean (nullable = true)
 |-- time: timestamp (nullable = true)
```

[9]:
```
## If you want to select one or a set of columns, we can use the select method
data.select('time').show(5)
```

```
+-------------------+
|               time|
+-------------------+
|1929-10-20 00:00:00|
|1929-12-18 00:00:00|
|1931-04-24 00:00:00|
|1931-06-23 00:00:00|
```

```
|1931-03-02 00:00:00|
+-------------------+
only showing top 5 rows
```

[10]:
```
data.select(['time', 'tornado']).show(5)
```

```
+-------------------+-------+
|               time|tornado|
+-------------------+-------+
|1929-10-20 00:00:00|  false|
|1929-12-18 00:00:00|  false|
|1931-04-24 00:00:00|  false|
|1931-06-23 00:00:00|  false|
|1931-03-02 00:00:00|  false|
+-------------------+-------+
only showing top 5 rows
```

[11]:
```
## If you want any row, we can take
data.take(1)
```

[11]:
```
[Row(station_number=39730, wban_number=99999, year=1929, month=10, day=20,
mean_temp=52.79999923706055, num_mean_temp_samples=4, mean_dew_point=45.5,
num_mean_dew_point_samples=4, mean_sealevel_pressure=None,
num_mean_sealevel_pressure_samples=None, mean_station_pressure=None,
num_mean_station_pressure_samples=None, mean_visibility=6.199999809265137,
num_mean_visibility_samples=4, mean_wind_speed=21.200000762939453,
num_mean_wind_speed_samples=4, max_sustained_wind_speed=29.899999618530273,
max_gust_wind_speed=None, max_temperature=50.0, max_temperature_explicit=False,
min_temperature=None, min_temperature_explicit=None, total_precipitation=0.0,
snow_depth=None, fog=False, rain=False, snow=False, hail=False, thunder=False,
tornado=False, time=datetime.datetime(1929, 10, 20, 0, 0))]
```

Each Row of the DataFrame is a `Row` which is similar to a dictionary, you can reference each element of the Row using the key. Now, also notice that the output of `take` is a list, so you need to index the list first

[12]:
```
## let's get the station_number only
data.take(1)[0]['station_number']
```

[12]:
```
39730
```

[13]:
```
## Let's check now how to filter data using another weather station data
stations = spark.read.format('bigquery') \
  .option('table', 'bigquery-public-data:noaa_gsod.stations') \
  .load()
```

[14]:
```
## Let's filter only US based stations
# Let's filter for just the US since this is a US based dataset
stations_us = stations.filter(F.col('Country')=='US')

print(f'Total stations are {stations.count()}, total US stations are␣
↪{stations_us.count()}')
```

```
Total stations are 29590, total US stations are 7161
```

[15]:
```
%%time
## Finally, we can describe our dataset using the describe command, similar to␣
↪Pandas
## Let's select just a few columns
keep_columns = ['station_number', 'mean_temp', 'thunder',␣
↪'mean_sealevel_pressure']
data.select(keep_columns).describe().show()
```

```
[Stage 18:============================>                           (2 + 2) / 4]

+-------+------------------+------------------+----------------------+
|summary|    station_number|         mean_temp|mean_sealevel_pressure|
+-------+------------------+------------------+----------------------+
|  count|         114420316|         114420316|              86731897|
|   mean| 507199.9578684261|52.122209996445854|    1014.8442087525018|
| stddev|298384.12645319354|24.222342560059765|      9.38153057246377|
|    min|              8209|            -118.5|                 900.0|
|    max|            999999|             110.0|     1079.699951171875|
+-------+------------------+------------------+----------------------+

CPU times: user 11.1 ms, sys: 1.66 ms, total: 12.8 ms
Wall time: 18.9 s
```

2 Data Types

I highly recommend reading this article Format Wars which covered the characteristics, structure, and differences between raw text, sequence, Avro, Parquet, and ORC data serializations.

There were several points discussed:

- Human Readable
- Row vs Column Oriented
- Read vs Write performance
- Appendable
- Splittable
- Metadata storage

We have 4 data types below

- Compressed CSV
- Parquet
- Avro
- CSV

Of these 3 are row oriented and 1 is column oriented. We have over 100M rows and 31 columns. Columnar compression should do fairly well in this scenerio.

[4]:
```
# Access staging bucket and see whats there
import os
GCS_LOCATION = os.getenv('DATA_BUCKET')
GCS_LOCATION
```

```
STAGING_BUCKET location: gs://dataproc-staging-us-
central1-1077780374322-lwjldfuu/
```

[ ]:
```
%%time
!gsutil rm -r {GCS_LOCATION}datagzip
data.write.option("compression","gzip").csv(f'{GCS_LOCATION}datagzip')
!gsutil du -sh {GCS_LOCATION}datagzip/*
```

```
Removing gs://dataproc-staging-us-
central1-1077780374322-lwjldfuu/datagzip/#1655740918347358…
/ [1 objects]
Operation completed over 1 objects.


3.08 GiB     gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/datagzip/*
CPU times: user 179 ms, sys: 62.5 ms, total: 242 ms
Wall time: 9min 53s
```

[ ]:
```
%%time
!gsutil rm -r {GCS_LOCATION}dataparquet
data.write.format("parquet").save(f'{GCS_LOCATION}dataparquet')
!gsutil du -sh {GCS_LOCATION}dataparquet/*
```

```
Removing gs://dataproc-staging-us-
central1-1077780374322-lwjldfuu/dataparquet/#1655741566985968…
Removing gs://dataproc-staging-us-
central1-1077780374322-lwjldfuu/dataparquet/_SUCCESS#1655741567151993…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataparquet/pa
rt-00000-ecbf1a52-7da7-4857-9fdd-f48c9c03610f-c000.snappy.parquet#16557415654079
91…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataparquet/pa
rt-00001-ecbf1a52-7da7-4857-9fdd-f48c9c03610f-c000.snappy.parquet#16557415652745
01…
/ [4 objects]
==> NOTE: You are performing a sequence of gsutil operations that may
run significantly faster if you instead use gsutil -m rm … Please
```

```
see the -m section under "gsutil help options" for further information
about when gsutil -m can be advantageous.

Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataparquet/pa
rt-00002-ecbf1a52-7da7-4857-9fdd-f48c9c03610f-c000.snappy.parquet#16557415665820
27…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataparquet/pa
rt-00003-ecbf1a52-7da7-4857-9fdd-f48c9c03610f-c000.snappy.parquet#16557415616279
49…
/ [6 objects]
Operation completed over 6 objects.


1.72 GiB     gs://dataproc-staging-us-
central1-1077780374322-lwjldfuu/dataparquet/*
CPU times: user 142 ms, sys: 40.8 ms, total: 183 ms
Wall time: 3min 54s
```

[ ]:
```
%%time
!gsutil rm -r {GCS_LOCATION}dataavro
data.write.format("avro").save(f'{GCS_LOCATION}dataavro')
!gsutil du -sh {GCS_LOCATION}dataavro/*
```

```
Removing gs://dataproc-staging-us-
central1-1077780374322-lwjldfuu/dataavro/#1655742062236332…
Removing gs://dataproc-staging-us-
central1-1077780374322-lwjldfuu/dataavro/_SUCCESS#1655742970570332…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataavro/part-
00000-2441faa1-1680-4894-920a-660b398f8048-c000.avro#1655742048661975…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataavro/part-
00000-76afd8a5-f31e-4b87-a48d-f6c7f22701ea-c000.avro#1655742548203079…
/ [4 objects]
==> NOTE: You are performing a sequence of gsutil operations that may
run significantly faster if you instead use gsutil -m rm … Please
see the -m section under "gsutil help options" for further information
about when gsutil -m can be advantageous.

Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataavro/part-
00001-2441faa1-1680-4894-920a-660b398f8048-c000.avro#1655742060703836…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataavro/part-
00001-76afd8a5-f31e-4b87-a48d-f6c7f22701ea-c000.avro#1655742549208485…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataavro/part-
00002-2441faa1-1680-4894-920a-660b398f8048-c000.avro#1655742061834562…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataavro/part-
00002-76afd8a5-f31e-4b87-a48d-f6c7f22701ea-c000.avro#1655742970078689…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataavro/part-
00003-2441faa1-1680-4894-920a-660b398f8048-c000.avro#1655742059267771…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataavro/part-
```

```
00003-76afd8a5-f31e-4b87-a48d-f6c7f22701ea-c000.avro#1655742959591683…
/ [10 objects]
Operation completed over 10 objects.


4.66 GiB     gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/dataavro/*
CPU times: user 179 ms, sys: 49 ms, total: 228 ms
Wall time: 7min 55s
```

[ ]:
```
%%time
!gsutil rm -r {GCS_LOCATION}datacsv
data.write.csv(f'{GCS_LOCATION}datacsv')
!gsutil du -sh {GCS_LOCATION}datacsv/*
```

```
Removing gs://dataproc-staging-us-
central1-1077780374322-lwjldfuu/datacsv/#1655742810311463…
Removing gs://dataproc-staging-us-
central1-1077780374322-lwjldfuu/datacsv/_SUCCESS#1655742810479233…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/datacsv/part-0
0000-184cd7ba-1147-494d-aee8-e8da0bb34a25-c000.csv#1655742431834302…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/datacsv/part-0
0001-184cd7ba-1147-494d-aee8-e8da0bb34a25-c000.csv#1655742442219329…
/ [4 objects]
==> NOTE: You are performing a sequence of gsutil operations that may
run significantly faster if you instead use gsutil -m rm … Please
see the -m section under "gsutil help options" for further information
about when gsutil -m can be advantageous.

Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/datacsv/part-0
0002-184cd7ba-1147-494d-aee8-e8da0bb34a25-c000.csv#1655742809770265…
Removing gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/datacsv/part-0
0003-184cd7ba-1147-494d-aee8-e8da0bb34a25-c000.csv#1655742804954864…
/ [6 objects]
Operation completed over 6 objects.


22.18 GiB    gs://dataproc-staging-us-central1-1077780374322-lwjldfuu/datacsv/*
CPU times: user 172 ms, sys: 49.8 ms, total: 222 ms
Wall time: 6min 36s
```

2.1 Why do we care?

The compression of each data type matter when running different operations and computations, let's compare the 3

[21]:
```
## Create our dataframes
data_parquet = spark.read.parquet(f'{GCS_LOCATION}dataparquet')
data_csv = spark.read.csv(f'{GCS_LOCATION}datacsv')
```

```
data_avro = spark.read.format("avro").load(f'{GCS_LOCATION}/dataavro')
```

[22]:
```
%%time
data_parquet.count()
```

```
[Stage 11:=================================================>       (12 + 2) / 14]

CPU times: user 8.26 ms, sys: 0 ns, total: 8.26 ms
Wall time: 2.24 s
```

[22]: 114420316

[23]:
```
%%time
data_csv.count()
```

```
[Stage 14:======================================================>(177 + 1) / 178]

CPU times: user 136 ms, sys: 42.3 ms, total: 178 ms
Wall time: 1min 2s
```

[23]: 114420316

[24]:
```
%%time
data_avro.count()
```

```
[Stage 17:=======================================================> (37 + 1) / 38]

CPU times: user 67.6 ms, sys: 8.39 ms, total: 76 ms
Wall time: 2min
```

[24]: 114420316

- *What is the compression ratio for the parquet to csv file?* > We have 1.7G/21G = 0.081 or 8.1% of original size
- *Which serialization would query a column faster?* > Parquet has a columnar format therefore a column of data has faster access and only needs to grab a subset of data
- *Which types of columns do you think has the best compression for parquet?* > Columns with repeated content will have better compressions such as categorical columns will have very high compression ratios, especially if they're integers since parquet has enhanced compression for types with smaller storage requirements.
- *When should you use flat files vs other data formats?* > If you need human readable data or you have small data sets. Interoperability - for sharing with other teams. Don't send Bob in accounting a parquet file! Bob will try to open it in excel and he'll get an error, call IT, and IT will tell Bob to clear his cookies and restart his computer. Bob will not be impressed.

- *If we want to do analysis with lots of aggregations what serialization should we use?* > Parquet
- *Is there any downside to Parquet?* > Parquet is non-appendable (immutable) which means that if we have new data coming in we can't grow the dataset with parquet. Parquet datasets are typically used for batch analysis after the data has reached a final state, such as on a date roll-over.
- *If you had to partition data into days as new data comes in with aggregations happening at end of day how would you operationalize this?* > Data coming in for a day is streamed into an Avro file which handles appends seamlessly, then once the day has completed and a new partition for data is created a batch job can convert the avro file into a parquet file for the DS/Analyst team to query against.

3 Data Aggregation

Let's perform different aggregations using different methods and GroupBy. Don't worry! GroupBy from DataFrames is very different than RDDs.

[28]:
```
%%time
## Let's start with sorting
#data_parquet.sort("mean_temp").show()
#data_parquet.sort("mean_temp").select("mean_temp").show()
data_parquet.sort("mean_temp").select("mean_temp").filter(F.col("mean_temp").
 ↪isNotNull()).show()
```

```
[Stage 20:=================================================>       (12 + 2) / 14]

+------------------+
|         mean_temp|
+------------------+
|            -118.5|
|-118.30000305175781|
| -117.5999984741211|
| -117.0999984741211|
| -115.9000015258789|
|-115.80000305175781|
|            -115.5|
|            -115.5|
|-115.19999694824219|
| -115.0999984741211|
|            -115.0|
| -114.9000015258789|
| -114.9000015258789|
| -114.5999984741211|
| -114.0999984741211|
|            -114.0|
| -113.9000015258789|
| -113.5999984741211|
| -113.5999984741211|
```

```
| -113.4000015258789|
+--------------------+
only showing top 20 rows

CPU times: user 19.8 ms, sys: 490 µs, total: 20.3 ms
Wall time: 7.54 s
```

[29]:

```
%%time
## Let's compare with avro
data_avro.sort("mean_temp").select("mean_temp").filter(F.col("mean_temp").
 ↪isNotNull()).show()
```

```
[Stage 21:=======================================================> (37 + 1) / 38]

+--------------------+
|           mean_temp|
+--------------------+
|              -118.5|
|-118.30000305175781|
| -117.5999984741211|
| -117.0999984741211|
| -115.9000015258789|
|-115.80000305175781|
|              -115.5|
|              -115.5|
|-115.19999694824219|
| -115.0999984741211|
|              -115.0|
| -114.9000015258789|
| -114.9000015258789|
| -114.5999984741211|
| -114.0999984741211|
|              -114.0|
| -113.9000015258789|
| -113.5999984741211|
| -113.5999984741211|
| -113.4000015258789|
+--------------------+
only showing top 20 rows

CPU times: user 63.2 ms, sys: 11.1 ms, total: 74.3 ms
Wall time: 2min 1s
```

[30]:

```
%%time
data_parquet.select(F.mean("mean_wind_speed").alias("Avg mean Wind")).show()
```

```
[Stage 22:===================================================>      (12 + 2) / 14]
```

```
+------------------+
|     Avg mean Wind|
+------------------+
|6.763403616672629|
+------------------+

CPU times: user 7.71 ms, sys: 4.93 ms, total: 12.6 ms
Wall time: 3.08 s
```

[31]:

```
%%time
data_parquet.select(F.max("mean_wind_speed")).show()
```

```
[Stage 25:===================================================>      (12 + 2) / 14]
```

```
+--------------------+
|max(mean_wind_speed)|
+--------------------+
|    96.9000015258789|
+--------------------+

CPU times: user 10.2 ms, sys: 1.07 ms, total: 11.3 ms
Wall time: 2.5 s
```

[32]:

```
%%time
data_parquet.select(F.min("mean_wind_speed")).show()
```

```
[Stage 28:===============================================>          (11 + 3) / 14]
```

```
+--------------------+
|min(mean_wind_speed)|
+--------------------+
|                 0.0|
+--------------------+

CPU times: user 9.27 ms, sys: 1.3 ms, total: 10.6 ms
Wall time: 2.41 s
```

[33]:

```
%%time
data_parquet.select(F.stddev("mean_wind_speed")).show()
```

```
[Stage 31:===================================================>      (12 + 2) / 14]
```

```
+----------------------------+
|stddev_samp(mean_wind_speed)|
```

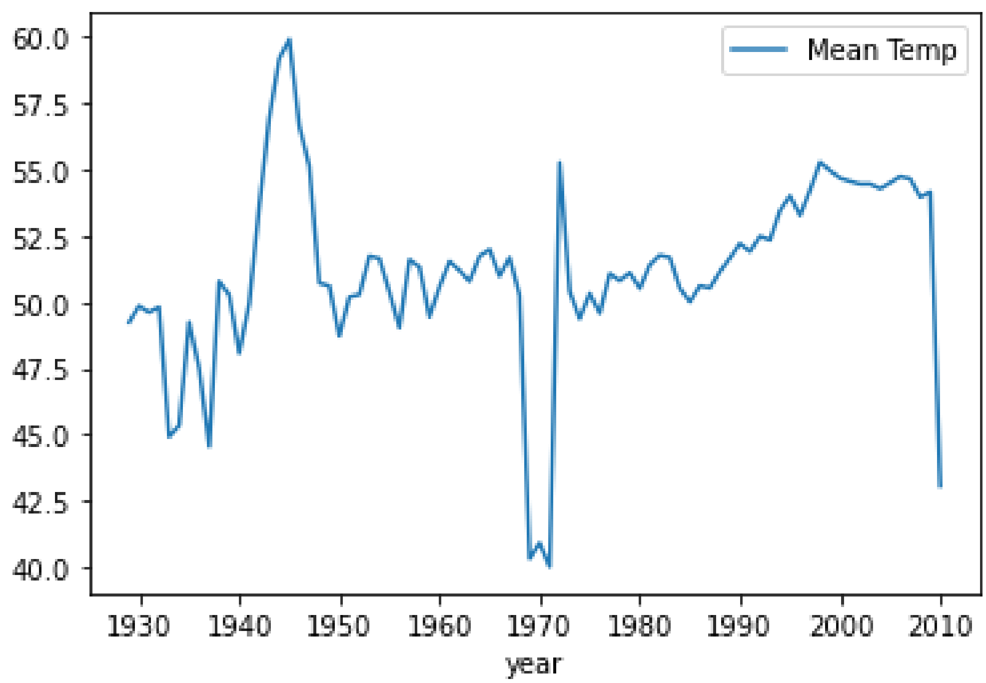
```
+----------------------------+
|             4.9270200129559|
+----------------------------+

CPU times: user 10.5 ms, sys: 1.9 ms, total: 12.4 ms
Wall time: 2.87 s
```

3.1 GroupBy

[34]:
```
%%time
## Let's suppose we want the average temperature by year
data_pandas = data_parquet.groupBy("year").agg(F.mean("mean_temp").alias("Mean␣
↪Temp")).toPandas()
data_pandas.sort_values("year").set_index("year").plot()
```

```
CPU times: user 2.38 s, sys: 163 ms, total: 2.54 s
Wall time: 10.3 s
```

[34]: <AxesSubplot:xlabel='year'>



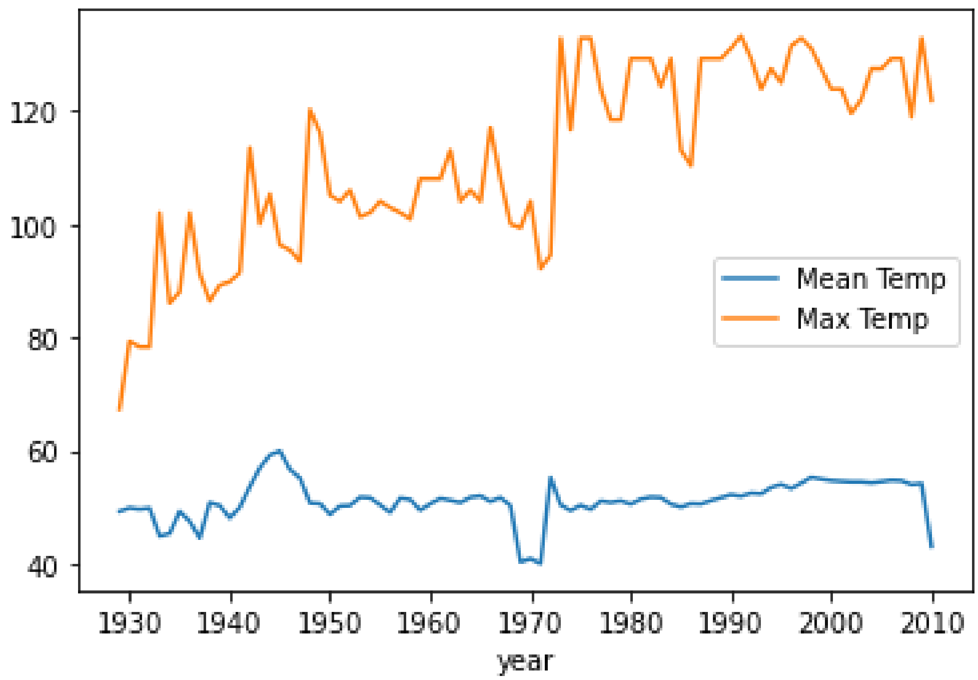
[35]:
```
data_parquet.groupBy("year").agg(F.mean("mean_temp").alias("Mean Temp")).count()
```

[35]: 82

[36]:
```
%%time
## Let's suppose we want more than one
data_pandas = data_parquet.groupBy("year").agg(F.mean("mean_temp").alias("Mean␣
↪Temp"),
                                                F.max("max_temperature").
↪alias("Max Temp")).toPandas()
data_pandas.sort_values("year").set_index("year").plot()
```

```
[Stage 43:=================================================>        (12 + 2) / 14]

CPU times: user 104 ms, sys: 34.1 ms, total: 138 ms
Wall time: 5.81 s
```

[36]: <AxesSubplot:xlabel='year'>

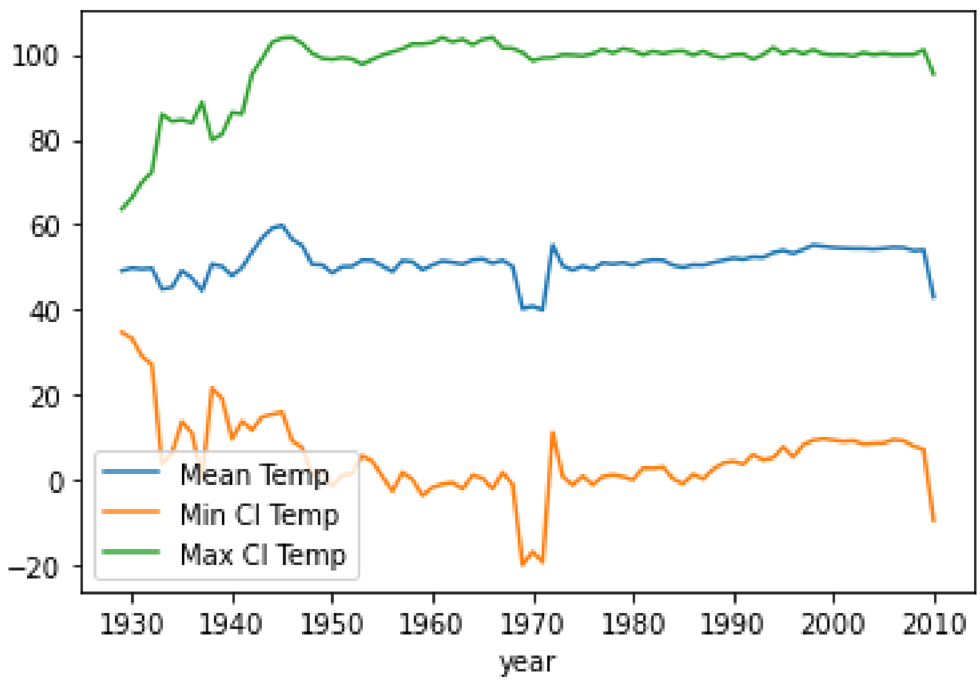


[37]:
```
%%time
## Let's suppose we want more than one
data_pandas = data_parquet.groupBy("year").agg(F.mean("mean_temp").alias("Mean␣
↪Temp"),
```

```
                                              F.stddev("mean_temp").alias("SD␣
↪Mean Temp")).toPandas()
data_pandas = data_pandas.sort_values("year").set_index("year")
data_pandas['Min CI Temp'] = data_pandas['Mean Temp'] - 2*data_pandas['SD Mean␣
↪Temp']
data_pandas['Max CI Temp'] = data_pandas['Mean Temp'] + 2*data_pandas['SD Mean␣
↪Temp']
del data_pandas['SD Mean Temp']
data_pandas.plot()
```

```
CPU times: user 130 ms, sys: 24.5 ms, total: 154 ms
Wall time: 5.27 s
```

[37]: <AxesSubplot:xlabel='year'>



[38]:
```
%%time
## Let's suppose we want more than one
data_parquet.groupBy(["year", 'month']).agg(F.mean("mean_temp").alias("Mean␣
↪Temp"),
                                              F.stddev("mean_temp").alias("SD␣
↪Mean Temp")).sort(['year','month']).show()
```

```
[Stage 49:==================================================>        (12 + 2) / 14]
```

```
+----+-----+------------------+------------------+
|year|month|         Mean Temp|      SD Mean Temp|
+----+-----+------------------+------------------+
|1929|    8|60.552419416366085|3.1990462574283702|
|1929|    9| 61.32605038170053| 4.033576425771579|
|1929|   10| 50.62124188117732| 4.801964157921469|
|1929|   11|47.099176249747615| 6.103062771817114|
|1929|   12| 45.37819060909536| 6.024023504130773|
|1930|    1|  44.1638498261501| 4.924192609845936|
|1930|    2| 39.85389948800284| 5.424807085849544|
|1930|    3|43.270230277588496|  6.20364872795214|
|1930|    4| 46.59947278336188| 4.285578098496348|
|1930|    5| 51.23745814294719|5.8525031165383705|
|1930|    6|57.698947384482935| 4.381964851854148|
|1930|    7| 58.29831368123098|3.4151141395515636|
|1930|    8| 58.91847454006389| 4.035375986521116|
|1930|    9| 57.15355978042948|  5.05659012503392|
|1930|   10|51.519968809464046| 4.618773602848772|
|1930|   11| 46.13617694547391| 6.240501759905245|
|1930|   12|44.314175671899264|  6.18088579805713|
|1931|    1|40.092450479469676| 7.572812511108157|
|1931|    2|39.418145178466716| 7.659416160405766|
|1931|    3| 40.05160875578184| 8.428746256340379|
+----+-----+------------------+------------------+
only showing top 20 rows

CPU times: user 9.31 ms, sys: 7.83 ms, total: 17.1 ms
Wall time: 6.19 s
```

3.2 User Defined Functions

[39]:
```
## Let's recall how we created the time column from before
data_parquet_time = data_parquet.withColumn("time",
                                F.concat(F.col("year"),
                                F.lit("-"), F.col("month"),
                                F.lit("-"), F.col("day")) \
                                .cast(types.TimestampType()))
```

[40]:
```
%%time
data_parquet_time.select('time').show(5)
```

```
+-------------------+
|               time|
+-------------------+
|1929-10-20 00:00:00|
|1929-12-18 00:00:00|
```

```
|1931-04-24 00:00:00|
|1931-06-23 00:00:00|
|1931-03-02 00:00:00|
+-------------------+
only showing top 5 rows

CPU times: user 3.42 ms, sys: 576 µs, total: 4 ms
Wall time: 483 ms
```

[41]:
```
## Can we do it differently? Yes! UDF. You can create UDF that will work row by␣
↪row in your dataframe
def create_date_from_parts(year, month, day):
    return f'{year}-{month}-{day}'

create_date_udf = F.udf(create_date_from_parts, types.StringType())
data_parquet_time_udf = data_parquet.withColumn("time", create_date_udf('year',␣
↪'month', 'day').cast(types.TimestampType()))
```

[42]:
```
%%time
data_parquet_time_udf.select('time').show(5)
```

```
[Stage 53:>                                                          (0 + 1) / 1]

+-------------------+
|               time|
+-------------------+
|1929-10-20 00:00:00|
|1929-12-18 00:00:00|
|1931-04-24 00:00:00|
|1931-06-23 00:00:00|
|1931-03-02 00:00:00|
+-------------------+
only showing top 5 rows

CPU times: user 5.98 ms, sys: 0 ns, total: 5.98 ms
Wall time: 1.49 s
```

UDFs are typically much slower than built-in Spark functionality. The reason for this is becauase they have to serialize and deserialize the data for every row that the function is applied to. There have been recent improvements to UDF for some analytical results with Pandas UDFs that return scalars or groupby maps. Some more information about why UDFs are inefficent can be found here https://blog.cloudera.com/blog/2017/02/working-with-udfs-in-apache-spark/

[43]:
```
%%time
## Let's look at other examples
from pyspark.sql.functions import udf
@udf("double")
```

```python
def squared_udf(s):
    return s * s


data_udf = data_parquet.withColumn("square_temp", squared_udf(F.
 ↪col("mean_temp")))
data_udf.select("square_temp").show()
```

```
+------------------+
|       square_temp|
+------------------+
|2787.8399194335943|
|           2256.25|
|2520.0400765991217|
| 4238.009801330569|
|1831.8399346923834|
|            4489.0|
| 4678.560208740237|
| 4108.809804382327|
|1689.2098745727562|
| 605.1600187683107|
|            5041.0|
|            5041.0|
| 3047.040084228516|
| 3058.089915618897|
|             121.0|
| 3931.290095672608|
|           4970.25|
|              25.0|
|106.09000392913822|
|            4225.0|
+------------------+
only showing top 20 rows

CPU times: user 9.54 ms, sys: 3.68 ms, total: 13.2 ms
Wall time: 741 ms
```

[44]:
```python
%%time
data_no_udf = data_parquet.withColumn("square_temp", F.col("mean_temp")**2)
data_udf.select("square_temp").show()
```

```
+------------------+
|       square_temp|
+------------------+
|2787.8399194335943|
|           2256.25|
|2520.0400765991217|
```

```
| 4238.009801330569|
|1831.8399346923834|
|            4489.0|
| 4678.560208740237|
| 4108.809804382327|
|1689.2098745727562|
| 605.1600187683107|
|            5041.0|
|            5041.0|
| 3047.040084228516|
| 3058.089915618897|
|             121.0|
| 3931.290095672608|
|           4970.25|
|              25.0|
|106.09000392913822|
|            4225.0|
+------------------+
only showing top 20 rows

CPU times: user 5.16 ms, sys: 0 ns, total: 5.16 ms
Wall time: 522 ms
```

[45]:

```
%%time
## You can also use UDF with select
data_parquet.select("mean_temp", squared_udf("mean_temp").
↪alias("squared_temp")).show()
```

```
+------------------+------------------+
|         mean_temp|      squared_temp|
+------------------+------------------+
|  52.79999923706055|2787.8399194335943|
|              47.5|           2256.25|
|  50.20000076293945|2520.0400765991217|
|   65.0999984741211| 4238.009801330569|
|  42.79999923706055|1831.8399346923834|
|              67.0|            4489.0|
|   68.4000015258789| 4678.560208740237|
|   64.0999984741211| 4108.809804382327|
| 41.099998474121094|1689.2098745727562|
| 24.600000381469727| 605.1600187683107|
|              71.0|            5041.0|
|              71.0|            5041.0|
|  55.20000076293945| 3047.040084228516|
|  55.29999923706055| 3058.089915618897|
|             -11.0|             121.0|
|  62.70000076293945| 3931.290095672608|
|              70.5|           4970.25|
```

```
|               -5.0|              25.0|
|-10.300000190734863|106.09000392913822|
|               65.0|            4225.0|
+-------------------+------------------+
only showing top 20 rows

CPU times: user 0 ns, sys: 5.41 ms, total: 5.41 ms
Wall time: 459 ms
```

4 Spark SQL

Finally, let's work with Spark SQL. Spark allows us to combine the power of SQL with Spark and the Dataframes API

[46]:
```
%%time
## Let's run an example
# First we need to create a temporary table that we can query
data_parquet.registerTempTable('data')
spark.sql(
"""
select mean_temp
from data
""").show()
```

```
+-------------------+
|          mean_temp|
+-------------------+
|  52.79999923706055|
|               47.5|
|  50.20000076293945|
|   65.0999984741211|
|  42.79999923706055|
|               67.0|
|   68.4000015258789|
|   64.0999984741211|
| 41.099998474121094|
| 24.600000381469727|
|               71.0|
|               71.0|
|  55.20000076293945|
|  55.29999923706055|
|              -11.0|
|  62.70000076293945|
|               70.5|
|               -5.0|
|-10.300000190734863|
|               65.0|
+-------------------+
```

```
only showing top 20 rows

CPU times: user 0 ns, sys: 4 ms, total: 4 ms
Wall time: 554 ms
```

[47]:
```
%%time
data_parquet.select("mean_temp").show()
```

```
+------------------+
|         mean_temp|
+------------------+
| 52.79999923706055|
|              47.5|
| 50.20000076293945|
|  65.0999984741211|
| 42.79999923706055|
|              67.0|
|  68.4000015258789|
|  64.0999984741211|
|41.099998474121094|
|24.600000381469727|
|              71.0|
|              71.0|
| 55.20000076293945|
| 55.29999923706055|
|             -11.0|
| 62.70000076293945|
|              70.5|
|              -5.0|
|-10.300000190734863|
|              65.0|
+------------------+
only showing top 20 rows

CPU times: user 0 ns, sys: 4.84 ms, total: 4.84 ms
Wall time: 369 ms
```

[48]:
```
## Let's run multiple querys similar to the ones we ran before
spark.sql(
"""
select mean_temp, power(mean_temp, 2) as squared_temp
from data
""").show()
```

```
+------------------+-----------------+
|         mean_temp|     squared_temp|
+------------------+-----------------+
| 52.79999923706055|2787.8399194335943|
```

```
|              47.5|          2256.25|
|  50.20000076293945|2520.0400765991217|
|   65.0999984741211| 4238.009801330569|
|  42.79999923706055|1831.8399346923834|
|              67.0|           4489.0|
|   68.4000015258789| 4678.560208740237|
|   64.0999984741211| 4108.809804382327|
| 41.099998474121094|1689.2098745727562|
| 24.600000381469727| 605.1600187683107|
|              71.0|           5041.0|
|              71.0|           5041.0|
|  55.20000076293945| 3047.040084228516|
|  55.29999923706055| 3058.089915618897|
|             -11.0|            121.0|
|  62.70000076293945| 3931.290095672608|
|              70.5|          4970.25|
|              -5.0|             25.0|
|-10.300000190734863|106.09000392913822|
|              65.0|           4225.0|
+-------------------+------------------+
only showing top 20 rows
```

[49]:

```
%%time
## Let's run multiple querys similar to the ones we ran before
spark.sql(
"""
select
    year,
    month,
    avg(mean_temp) as mean,
    std(mean_temp) as st_dev
from
    data
group by
    year,
    month
order by
    year,
    month
""").show()
```

```
[Stage 60:=================================================>       (12 + 2) / 14]

+----+-----+------------------+------------------+
|year|month|              mean|            st_dev|
+----+-----+------------------+------------------+
|1929|    8|60.552419416366085|3.1990462574283702|
```

```
|1929|    9| 61.32605038170053| 4.033576425771579|
|1929|   10| 50.62124188117732| 4.801964157921469|
|1929|   11|47.099176249747615| 6.103062771817114|
|1929|   12| 45.37819060909536| 6.024023504130773|
|1930|    1|  44.1638498261501| 4.924192609845936|
|1930|    2| 39.85389948800284| 5.424807085849544|
|1930|    3|43.270230277588496|  6.20364872795214|
|1930|    4| 46.59947278336188| 4.285578098496348|
|1930|    5| 51.23745814294719|5.852503116538370 5|
|1930|    6|57.698947384482935| 4.381964851854148|
|1930|    7| 58.29831368123098|3.4151141395515636|
|1930|    8| 58.91847454006389| 4.035375986521116|
|1930|    9| 57.15355978042948|  5.05659012503392|
|1930|   10|51.519968809464046| 4.618773602848772|
|1930|   11| 46.13617694547391| 6.240501759905245|
|1930|   12|44.314175671899264|  6.18088579805713|
|1931|    1|40.092450479469676| 7.572812511108157|
|1931|    2|39.418145178466716| 7.65941616040576 6|
|1931|    3| 40.05160875578184| 8.428746256340379|
+----+-----+------------------+------------------+
only showing top 20 rows

CPU times: user 11.2 ms, sys: 3.23 ms, total: 14.4 ms
Wall time: 6.03 s
```

[50]:
```python
## We can save the Spark SQL query as a dataframe
df_sql = spark.sql(
"""
select
    year,
    month,
    avg(mean_temp) as mean,
    std(mean_temp) as st_dev
from
    data
group by
    year,
    month
order by
    year,
    month
""")

df_sql_pd = df_sql.toPandas()
df_sql_pd = df_sql_pd.set_index(["year", 'month'])
df_sql_pd
```

[50]:
```
                 mean     st_dev
year month
1929 8      60.552419   3.199046
     9      61.326050   4.033576
     10     50.621242   4.801964
     11     47.099176   6.103063
     12     45.378191   6.024024
…                 …          …
2009 12     39.834150  26.892829
2010 1      37.385471  28.259400
     2      39.672241  27.192215
     3      46.851747  23.391840
     4      53.084282  20.170773

[969 rows x 2 columns]
```

[52]:
```python
%%time
## Let's also join dataframes
stations = spark.read.format('bigquery') \
  .option('table', 'bigquery-public-data:noaa_gsod.stations') \
  .load()

stations_us = stations.filter(F.col('Country')=='US')

## One of the dataframes is quite small, so let's broadcast it!
# join_data = data_parquet.join(F.broadcast(stations_us), stations_us.
 ↪usaf==data_parquet.station_number, 'inner')
join_data = data_parquet.join(stations_us, stations_us.usaf==data_parquet.
 ↪station_number, 'inner')
join_data.count()
```

```
[Stage 76:=================================================>       (12 + 2) / 14]

CPU times: user 17.1 ms, sys: 4.96 ms, total: 22 ms
Wall time: 19.2 s
```

[52]: 4584888375

[53]:
```python
join_data.printSchema()
```

```
root
 |-- station_number: long (nullable = true)
 |-- wban_number: long (nullable = true)
 |-- year: long (nullable = true)
 |-- month: long (nullable = true)
```

```
 |-- day: long (nullable = true)
 |-- mean_temp: double (nullable = true)
 |-- num_mean_temp_samples: long (nullable = true)
 |-- mean_dew_point: double (nullable = true)
 |-- num_mean_dew_point_samples: long (nullable = true)
 |-- mean_sealevel_pressure: double (nullable = true)
 |-- num_mean_sealevel_pressure_samples: long (nullable = true)
 |-- mean_station_pressure: double (nullable = true)
 |-- num_mean_station_pressure_samples: long (nullable = true)
 |-- mean_visibility: double (nullable = true)
 |-- num_mean_visibility_samples: long (nullable = true)
 |-- mean_wind_speed: double (nullable = true)
 |-- num_mean_wind_speed_samples: long (nullable = true)
 |-- max_sustained_wind_speed: double (nullable = true)
 |-- max_gust_wind_speed: double (nullable = true)
 |-- max_temperature: double (nullable = true)
 |-- max_temperature_explicit: boolean (nullable = true)
 |-- min_temperature: double (nullable = true)
 |-- min_temperature_explicit: boolean (nullable = true)
 |-- total_precipitation: double (nullable = true)
 |-- snow_depth: double (nullable = true)
 |-- fog: boolean (nullable = true)
 |-- rain: boolean (nullable = true)
 |-- snow: boolean (nullable = true)
 |-- hail: boolean (nullable = true)
 |-- thunder: boolean (nullable = true)
 |-- tornado: boolean (nullable = true)
 |-- usaf: string (nullable = true)
 |-- wban: string (nullable = true)
 |-- name: string (nullable = true)
 |-- country: string (nullable = true)
 |-- state: string (nullable = true)
 |-- call: string (nullable = true)
 |-- lat: double (nullable = true)
 |-- lon: double (nullable = true)
 |-- elev: string (nullable = true)
 |-- begin: string (nullable = true)
 |-- end: string (nullable = true)
```

[54]:
```
%%time
## let's now use Spark SQL to query this data
join_data.registerTempTable('data')
spark.sql(
"""
select
    state,
```

```
    avg(mean_temp) as mean,
    avg(lat),
    avg(lon)
from
    data
where
    state in ('CA', 'TX', 'NY')
group by
    state
order by
    state
""").show()
```

```
[Stage 80:=====================================================>   (13 + 1) / 14]

+-----+------------------+------------------+-------------------+
|state|              mean|          avg(lat)|           avg(lon)|
+-----+------------------+------------------+-------------------+
|   CA|53.747206531956735| 35.89247422036925|-119.49590371314966|
|   NY| 53.66972249742855| 42.22919509172085| -76.01350616411186|
|   TX| 53.81268076536781|30.791588005852184| -98.54278344775695|
+-----+------------------+------------------+-------------------+

CPU times: user 27.7 ms, sys: 0 ns, total: 27.7 ms
Wall time: 43 s
```

[ ]: