Dataframe operations

Spark DataFrames allow operations similar to Pandas dataframes. We demonstrate some of those.

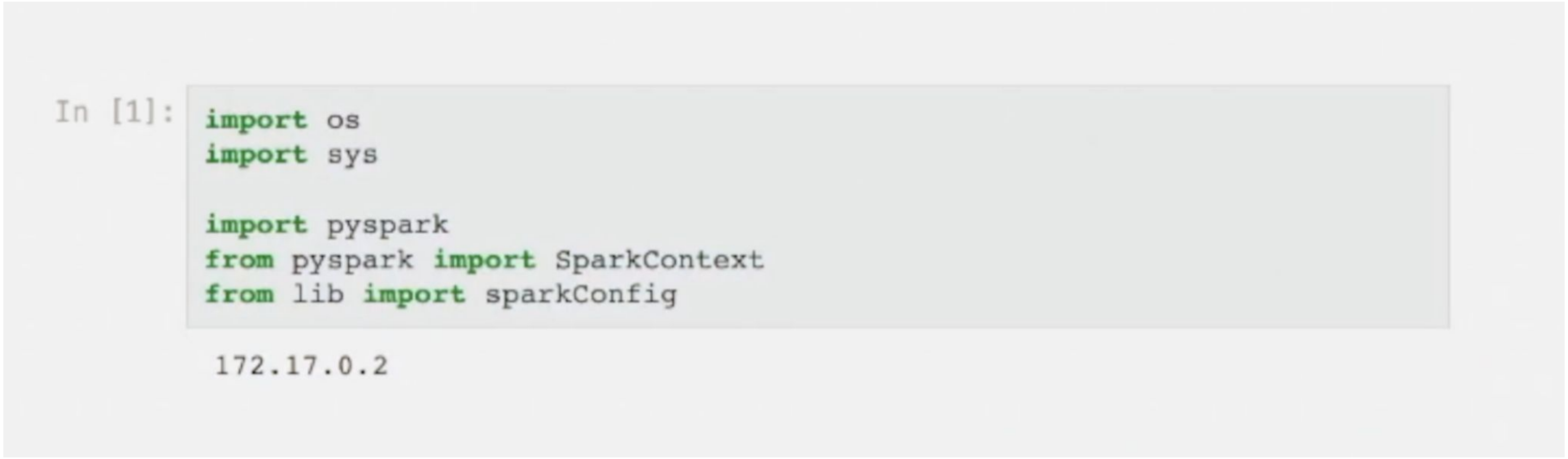
For more see the official guide and this article.

But first, some important details

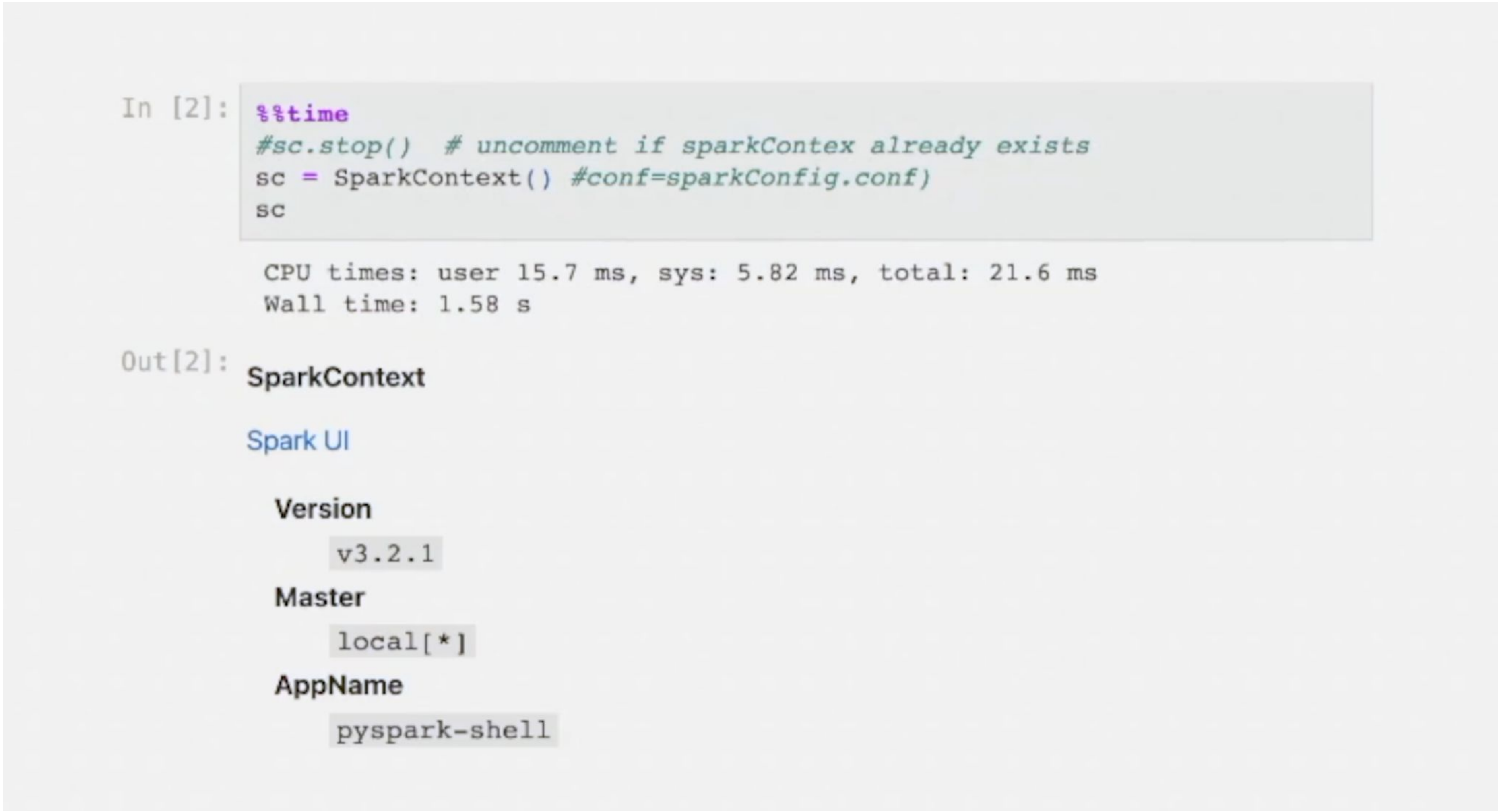
Spark and datahub can run in two modes:

- **Local mode** which means that it is run on the same computer as the head node. This convenient for small jobs and for debugging. Can also be done on your laptop.

- **Remote mode** in which the head node and the worker nodes are separate. This mode requires that the spark cluster is up and running. In this case the full resources of the clusters are available. This mode is useful when processing hard jobs.



```
In [1]: import os
        import sys

        import pyspark
        from pyspark import SparkContext
        from lib import sparkConfig

        172.17.0.2
```



```
In [2]: %%time
        #sc.stop()  # uncomment if sparkContex already exists
        sc = SparkContext() #conf=sparkConfig.conf)
        sc

        CPU times: user 15.7 ms, sys: 5.82 ms, total: 21.6 ms
        Wall time: 1.58 s

Out[2]:
```

SparkContext

Spark UI

Version

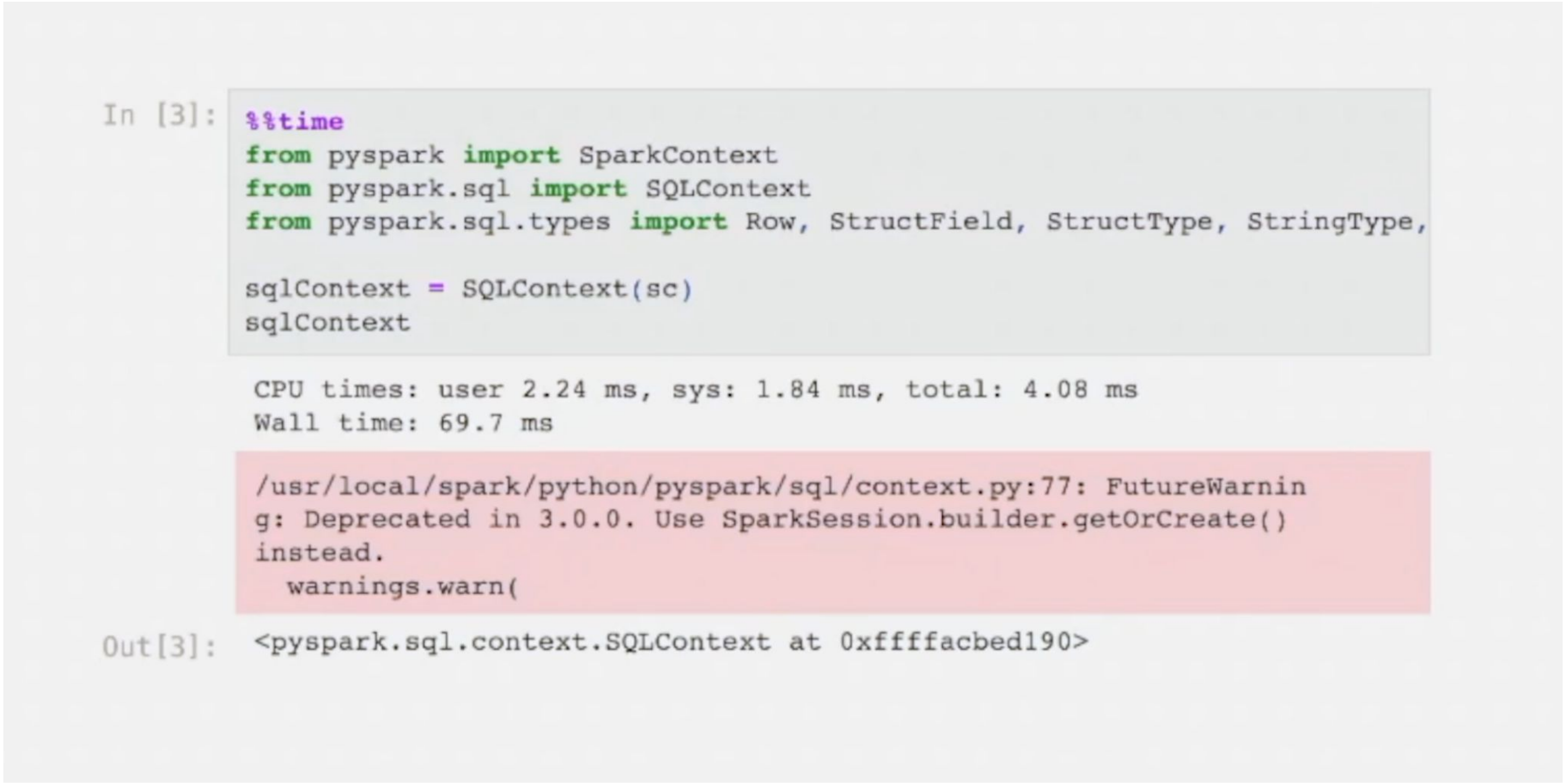
`v3.2.1`

Master

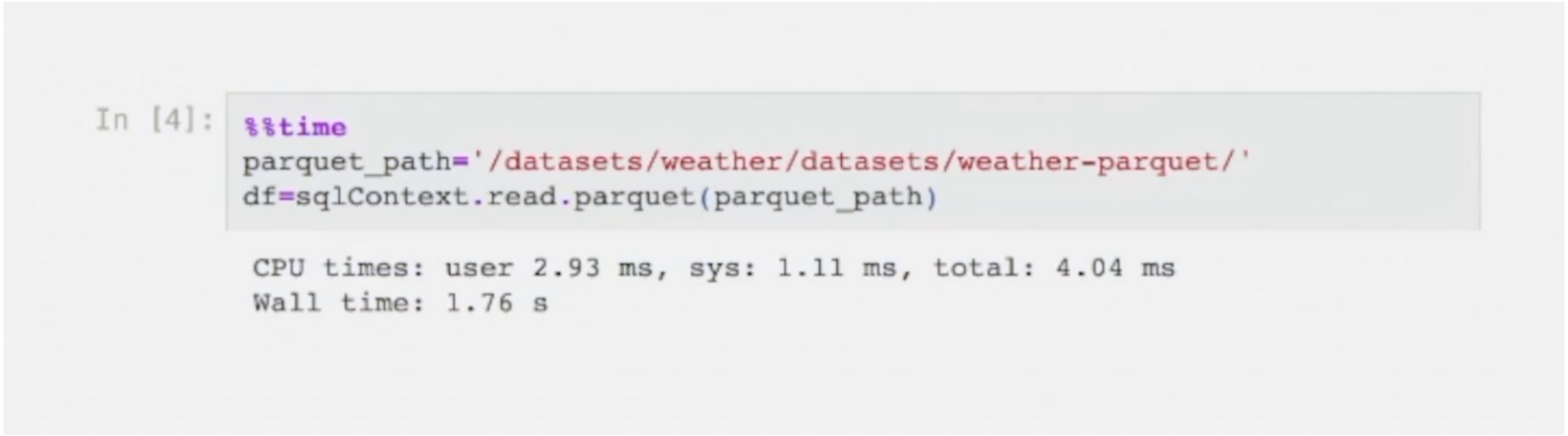
`local[*]`

AppName

`pyspark-shell`

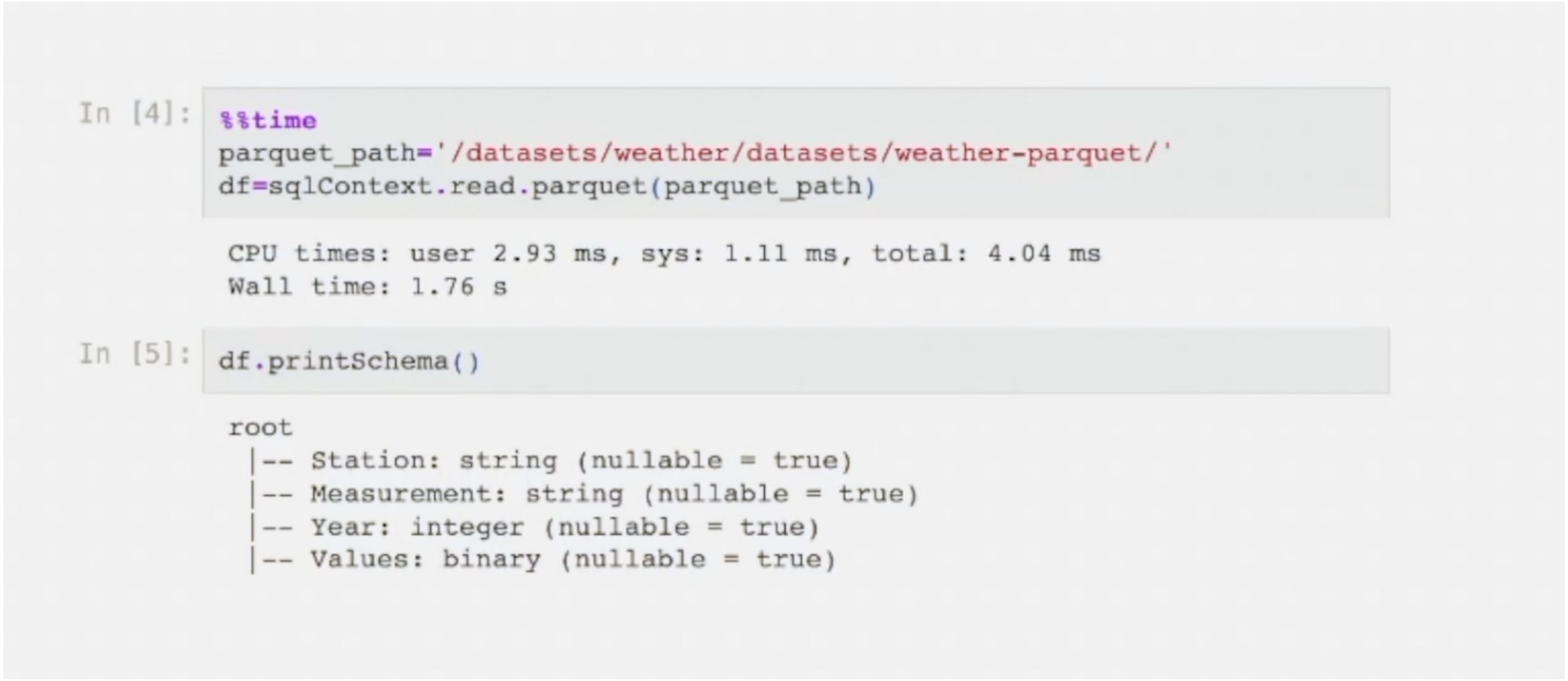


```
In [3]: %%time
        from pyspark import SparkContext
        from pyspark.sql import SQLContext
        from pyspark.sql.types import Row, StructField, StructType, StringType,

        sqlContext = SQLContext(sc)
        sqlContext

        CPU times: user 2.24 ms, sys: 1.84 ms, total: 4.08 ms
        Wall time: 69.7 ms

        /usr/local/spark/python/pyspark/sql/context.py:77: FutureWarnin
        g: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate()
        instead.
          warnings.warn(

Out[3]: <pyspark.sql.context.SQLContext at 0xffffacbed190>
```

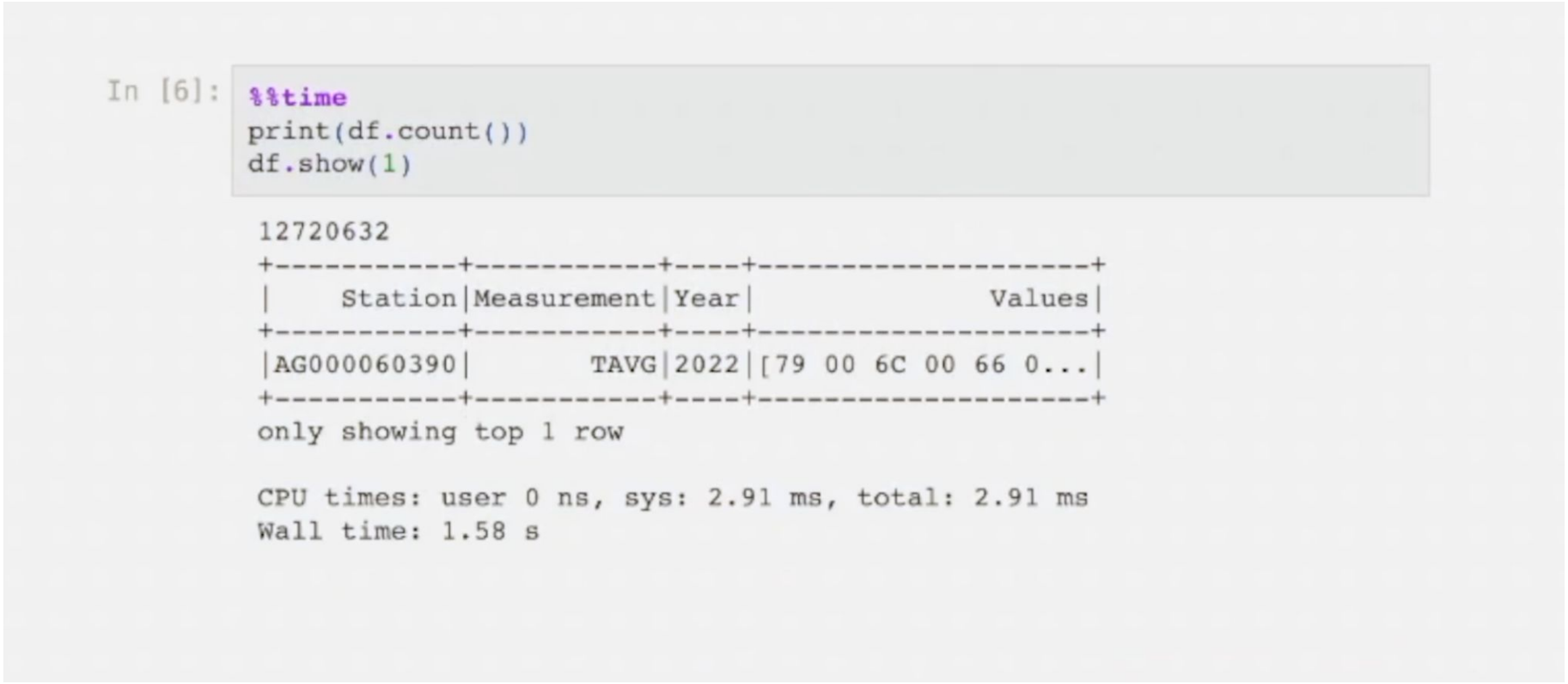


```
In [4]: %%time
parquet_path='/datasets/weather/datasets/weather-parquet/'
df=sqlContext.read.parquet(parquet_path)
```

```
CPU times: user 2.93 ms, sys: 1.11 ms, total: 4.04 ms
Wall time: 1.76 s
```



```
In [4]: %%time
        parquet_path='/datasets/weather/datasets/weather-parquet/'
        df=sqlContext.read.parquet(parquet_path)

        CPU times: user 2.93 ms, sys: 1.11 ms, total: 4.04 ms
        Wall time: 1.76 s

In [5]: df.printSchema()

        root
         |-- Station: string (nullable = true)
         |-- Measurement: string (nullable = true)
         |-- Year: integer (nullable = true)
         |-- Values: binary (nullable = true)
```



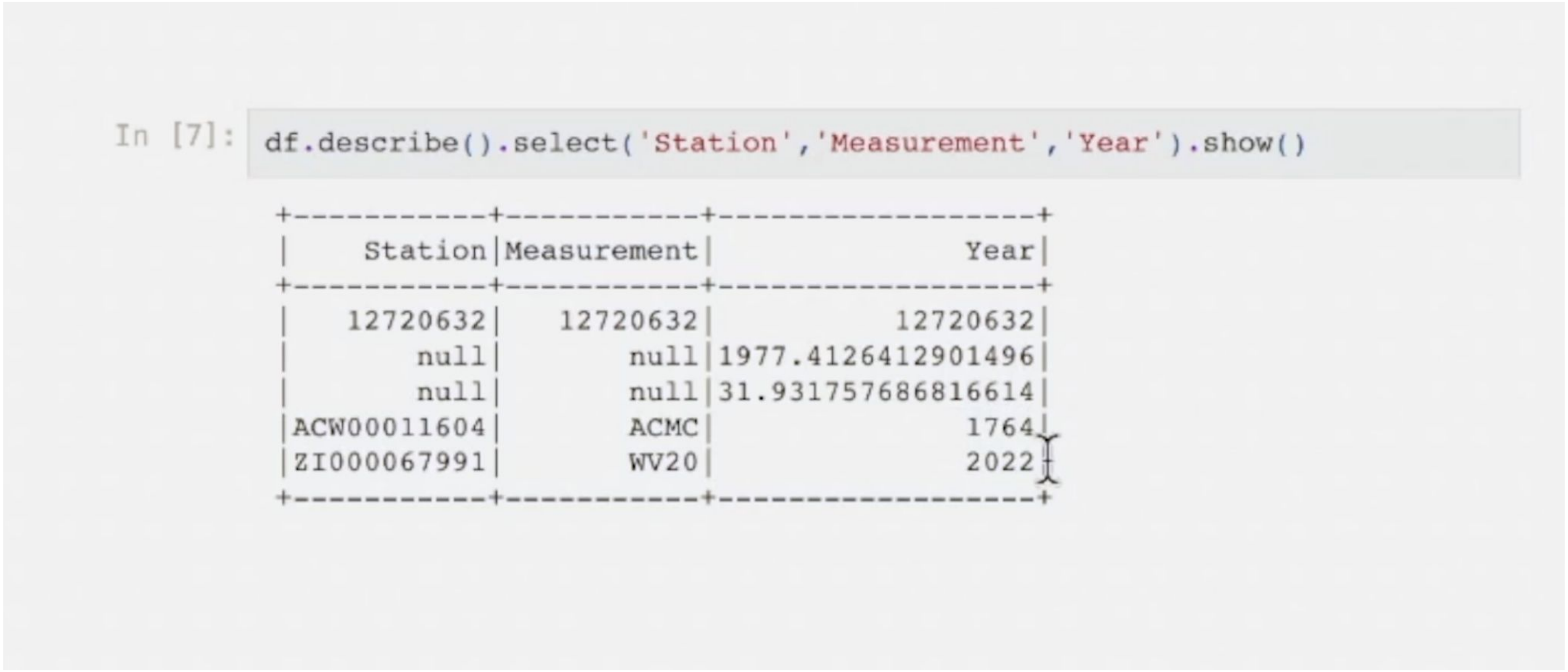
```
In [6]: %%time
        print(df.count())
        df.show(1)
```

```
12720632
+----------+-----------+----+--------------------+
|   Station|Measurement|Year|              Values|
+----------+-----------+----+--------------------+
|AG000060390|       TAVG|2022|[79 00 6C 00 66 0...|
+----------+-----------+----+--------------------+
only showing top 1 row

CPU times: user 0 ns, sys: 2.91 ms, total: 2.91 ms
Wall time: 1.58 s
```

.(describe)

The method `df.describe()` computers five statistics for each column of the dataframe `df.`

The statistics are **count**,**mean**,**std**,**min**,**max**



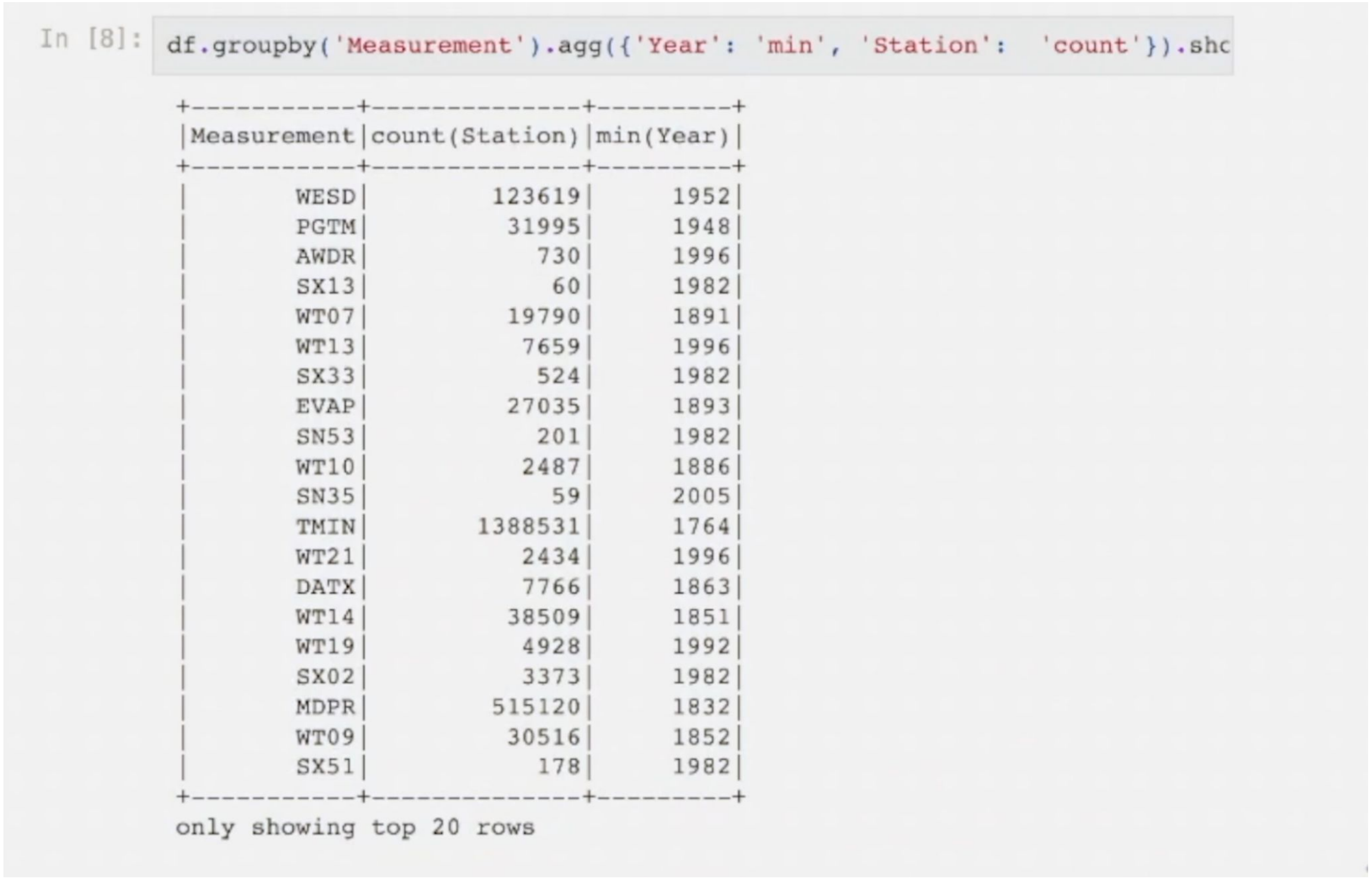
```
In [7]: df.describe().select('Station','Measurement','Year').show()
```

```
+-----------+-----------+------------------+
|    Station|Measurement|              Year|
+-----------+-----------+------------------+
|   12720632|   12720632|          12720632|
|       null|       null|1977.4126412901496|
|       null|       null| 31.931757686816614|
|ACW00011604|       ACMC|              1764|
|ZI000067991|       WV20|              2022|
+-----------+-----------+------------------+
```

groupby and agg

The method `groupby(col)` groups rows according the value of the column `(col)`.

The method `.agg(spec)` computes a summary for each group specified in `spec`



```
In [8]: df.groupby('Measurement').agg({'Year': 'min', 'Station':  'count'}).sho
```

```
+-----------+--------------+---------+
|Measurement|count(Station)|min(Year)|
+-----------+--------------+---------+
|       WESD|        123619|     1952|
|       PGTM|         31995|     1948|
|       AWDR|           730|     1996|
|       SX13|            60|     1982|
|       WT07|         19790|     1891|
|       WT13|          7659|     1996|
|       SX33|           524|     1982|
|       EVAP|         27035|     1893|
|       SN53|           201|     1982|
|       WT10|          2487|     1886|
|       SN35|            59|     2005|
|       TMIN|       1388531|     1764|
|       WT21|          2434|     1996|
|       DATX|          7766|     1863|
|       WT14|         38509|     1851|
|       WT19|          4928|     1992|
|       SX02|          3373|     1982|
|       MDPR|        515120|     1832|
|       WT09|         30516|     1852|
|       SX51|           178|     1982|
+-----------+--------------+---------+
only showing top 20 rows
```

Using SQL Queries on DataFrames

There are two main ways to manipulate DataFrames:

Imperative Manipulation

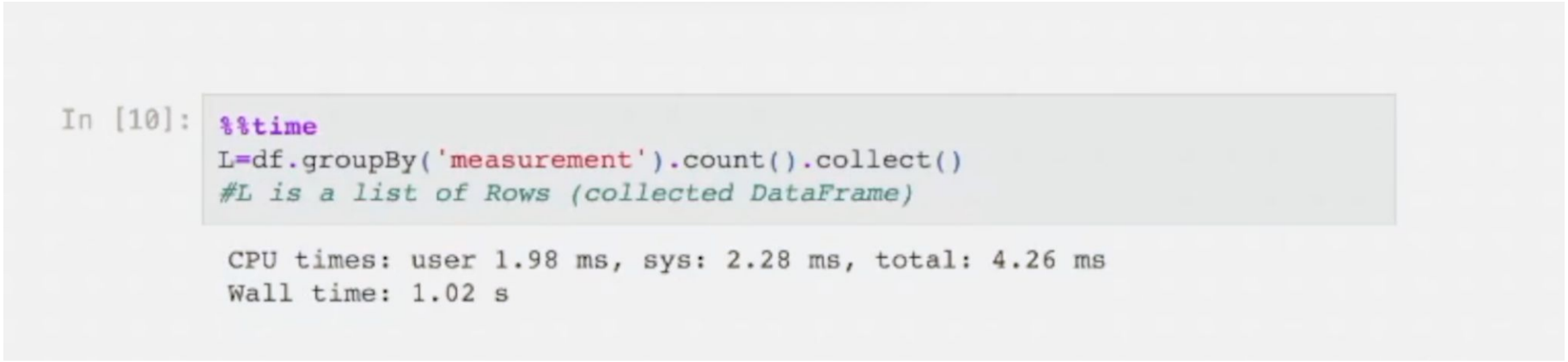
Using python methods such as `.select` and `.groupby`.

- Advantage: order of operations is specified
- Disadvantage: you need to describe both **what** is the result you want and **how** to get it.

Declarative Manipulation (SQL)

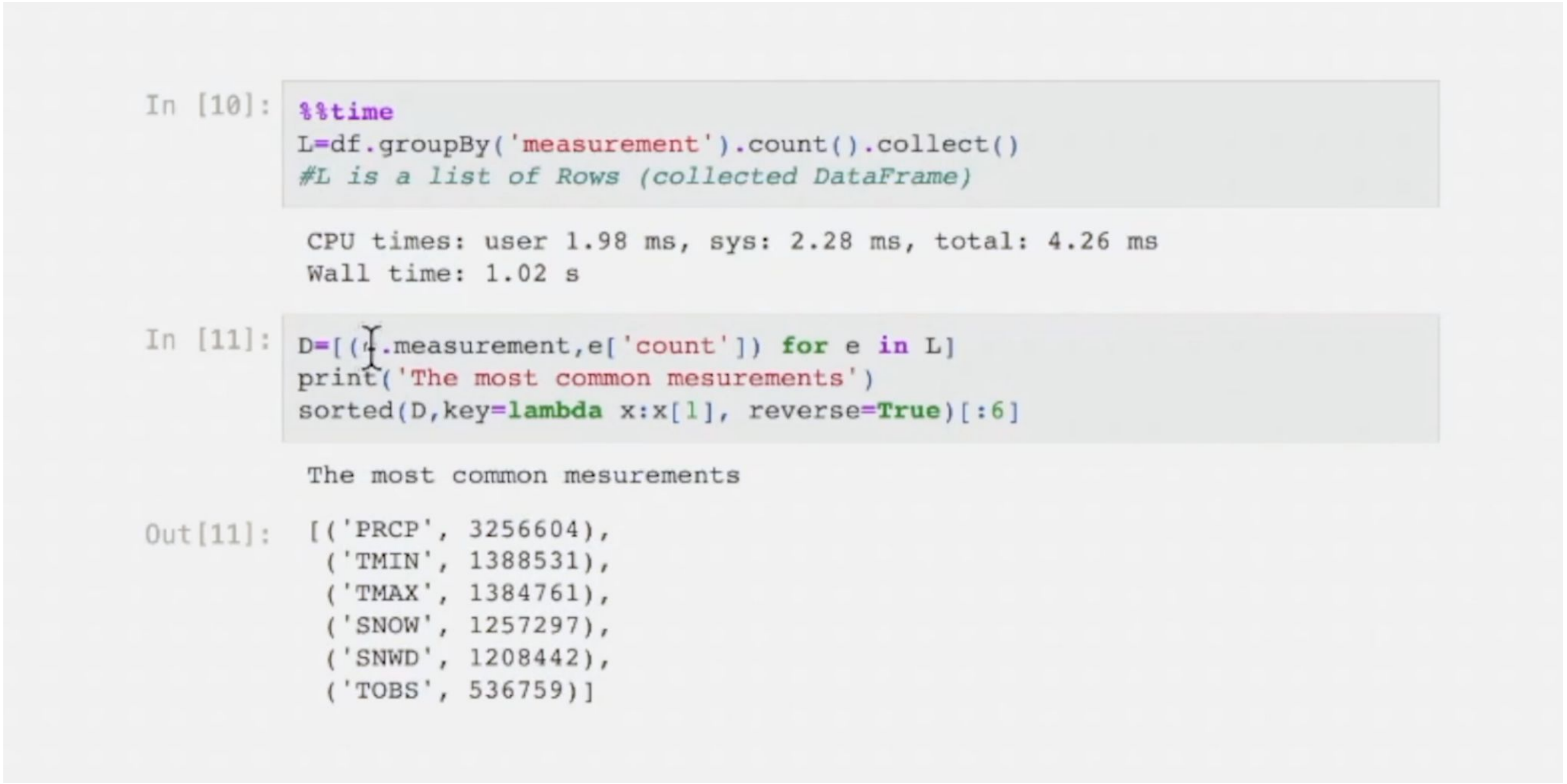
- Advantage: You need to describe only **what** is the result you want
- Disadvantage: SQL does not have primitives for common analysis operations such as **covariance**

Counting the number of occurrences of each measurement, comparatively

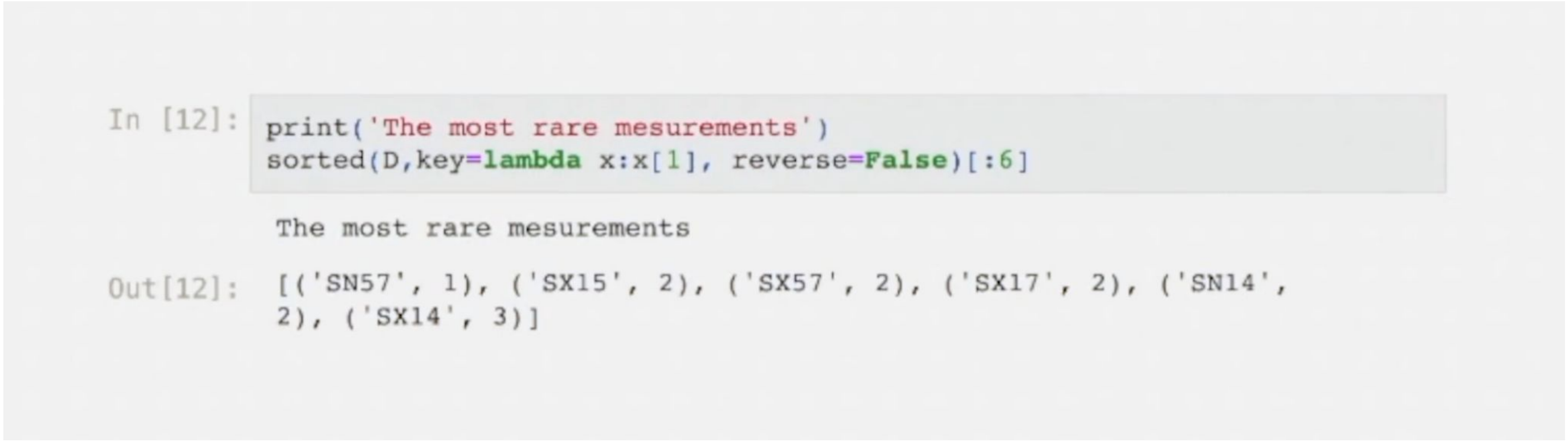


```
In [10]: %%time
         L=df.groupBy('measurement').count().collect()
         #L is a list of Rows (collected DataFrame)
```

```
CPU times: user 1.98 ms, sys: 2.28 ms, total: 4.26 ms
Wall time: 1.02 s
```



```
In [10]: %%time
         L=df.groupBy('measurement').count().collect()
         #L is a list of Rows (collected DataFrame)

         CPU times: user 1.98 ms, sys: 2.28 ms, total: 4.26 ms
         Wall time: 1.02 s

In [11]: D=[(e.measurement,e['count']) for e in L]
         print('The most common mesurements')
         sorted(D,key=lambda x:x[1], reverse=True)[:6]

         The most common mesurements

Out[11]: [('PRCP', 3256604),
          ('TMIN', 1388531),
          ('TMAX', 1384761),
          ('SNOW', 1257297),
          ('SNWD', 1208442),
          ('TOBS', 536759)]
```

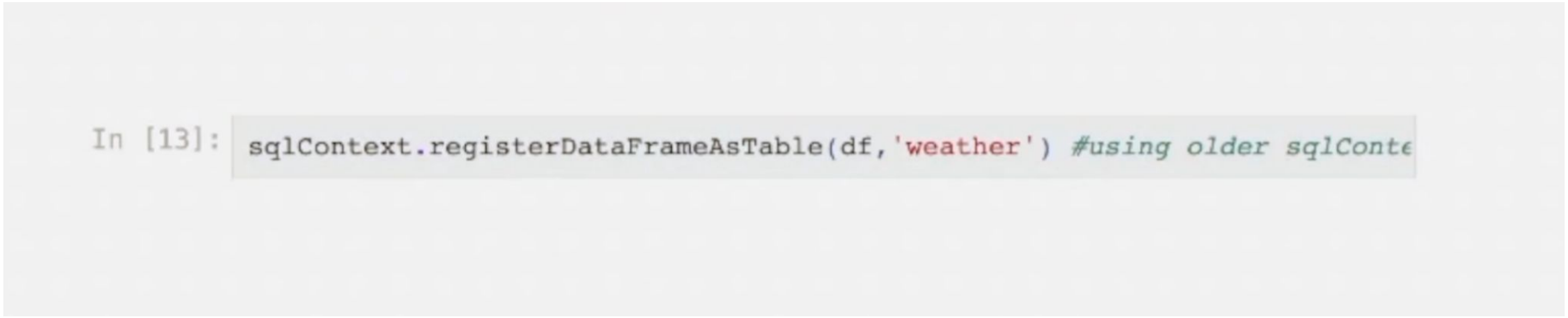


```
In [12]: print('The most rare mesurements')
         sorted(D,key=lambda x:x[1], reverse=False)[:6]

         The most rare mesurements

Out[12]: [('SN57', 1), ('SX15', 2), ('SX57', 2), ('SX17', 2), ('SN14',
         2), ('SX14', 3)]
```

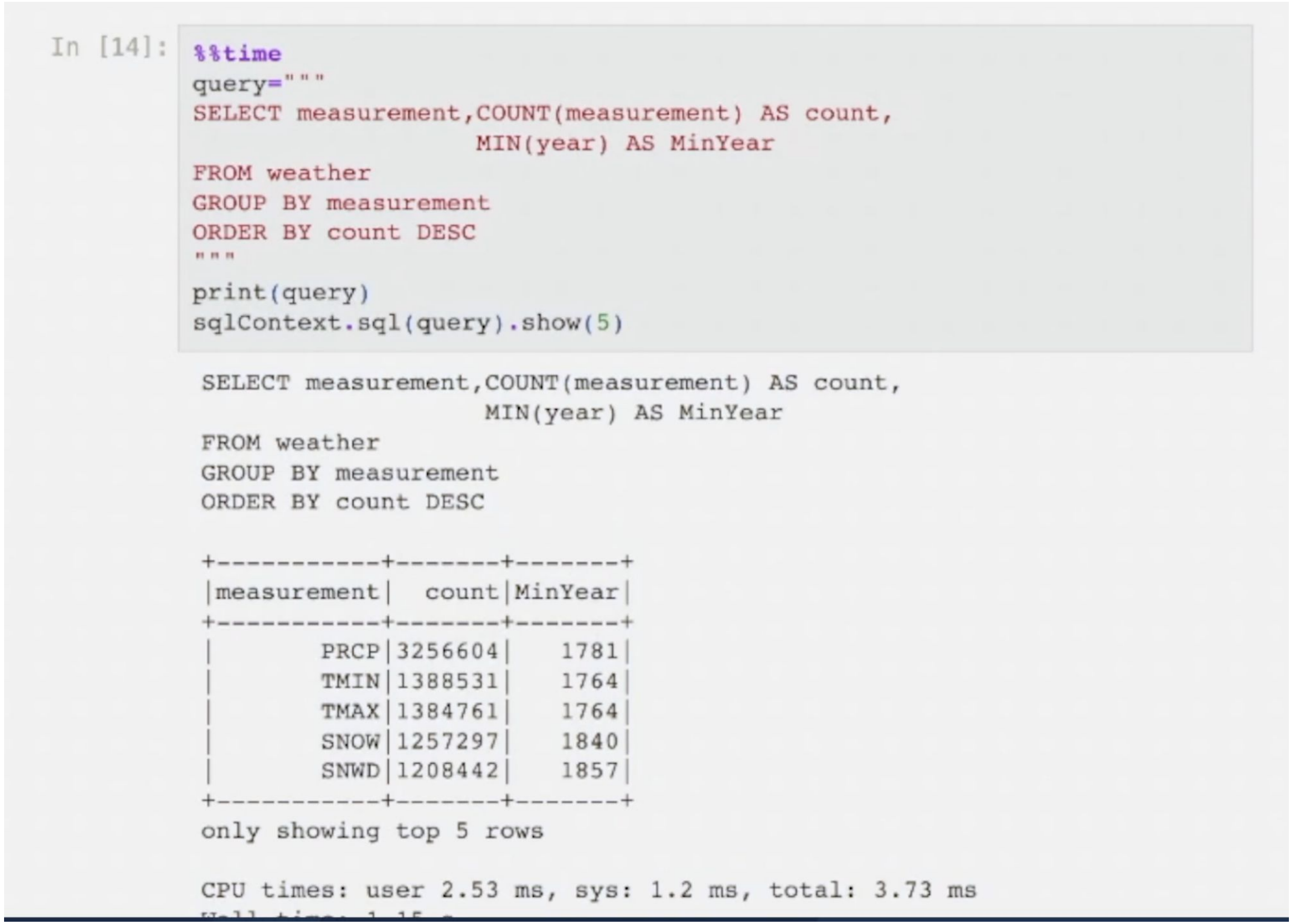
Counting the number of occurrences of each measurement, declaratively

Registering a dataframe as a table in a database.

In order to apply SQL commands to a dataframe, it has to first be registered as a table in the database managed by sqlContext.



```
In [13]: sqlContext.registerDataFrameAsTable(df,'weather') #using older sqlConte
```



```
In [14]: %%time
         query="""
         SELECT measurement,COUNT(measurement) AS count,
                            MIN(year) AS MinYear
         FROM weather
         GROUP BY measurement
         ORDER BY count DESC
         """
         print(query)
         sqlContext.sql(query).show(5)
```

```
SELECT measurement,COUNT(measurement) AS count,
                   MIN(year) AS MinYear
FROM weather
GROUP BY measurement
ORDER BY count DESC

+-----------+-------+-------+
|measurement|  count|MinYear|
+-----------+-------+-------+
|       PRCP|3256604|   1781|
|       TMIN|1388531|   1764|
|       TMAX|1384761|   1764|
|       SNOW|1257297|   1840|
|       SNWD|1208442|   1857|
+-----------+-------+-------+
only showing top 5 rows

CPU times: user 2.53 ms, sys: 1.2 ms, total: 3.73 ms
```

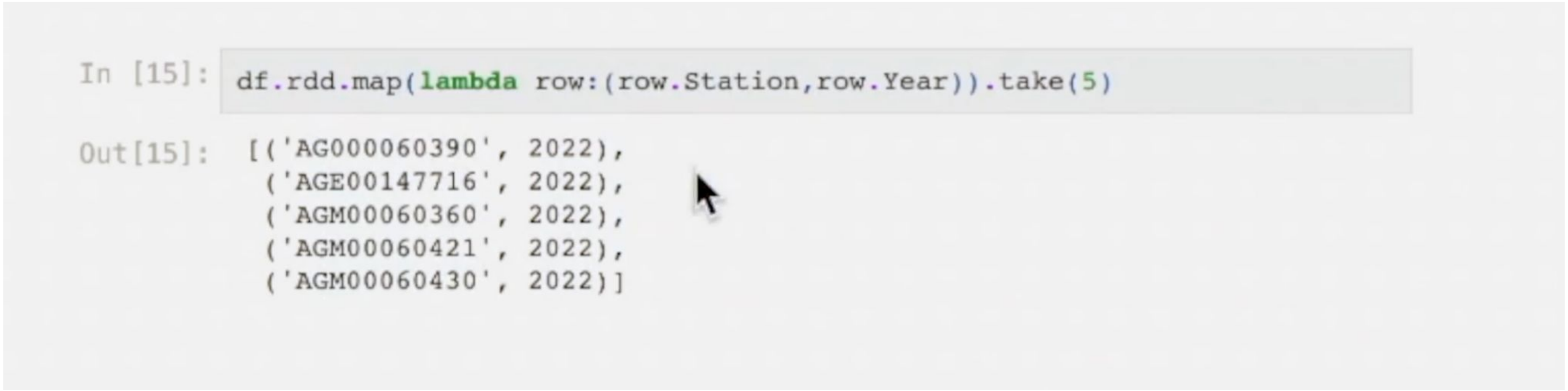
Performing a map command

- Dataframes do not support `map` and `reduce` operations.
- In order to perform a `map` or `reduce` on a dataframe, you first need to transform it into an RDD

Performing a map command

- Dataframes do not support `map` and `reduce` operations.
- In order to perform a `map` or `reduce` on a dataframe, you first need to transform it into an RDD

- This is a quick-and-dirty solution.
- A better way is to use built-in sparkSQL functions
- Or if you can’t find what you need, you can try and create a User-Defined-Function* (UDF)

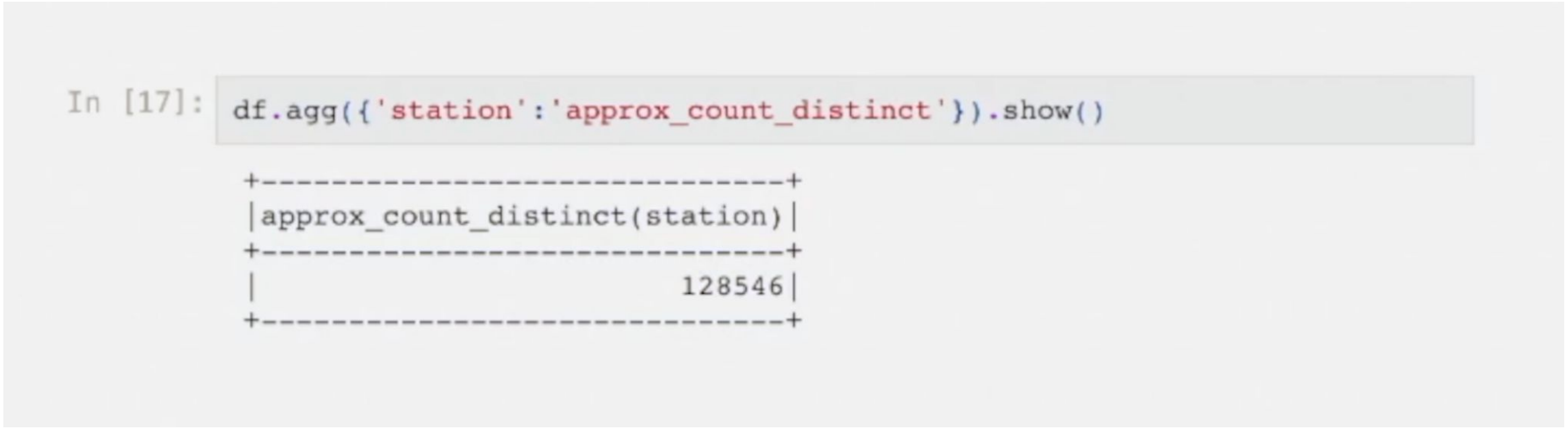


```
In [15]: df.rdd.map(lambda row:(row.Station,row.Year)).take(5)

Out[15]: [('AG000060390', 2022),
          ('AGE00147716', 2022),
          ('AGM00060360', 2022),
          ('AGM00060421', 2022),
          ('AGM00060430', 2022)]
```

Aggregations

- **Aggregation** can be used, in combination with built-in spark SQL functions, to compute statistics of a dataframe.
- Computation will be fast thanks to combined optimizations with database operations.

- A partial list: `count()`, `approx_count_distinct()`, `avg()`, `max()`, `min()`
- Of these, the interesting one is `approx_count_distinct()` which uses sampling to get an approximate count fast.



In [17]:

```
df.agg({'station':'approx_count_distinct'}).show()
```

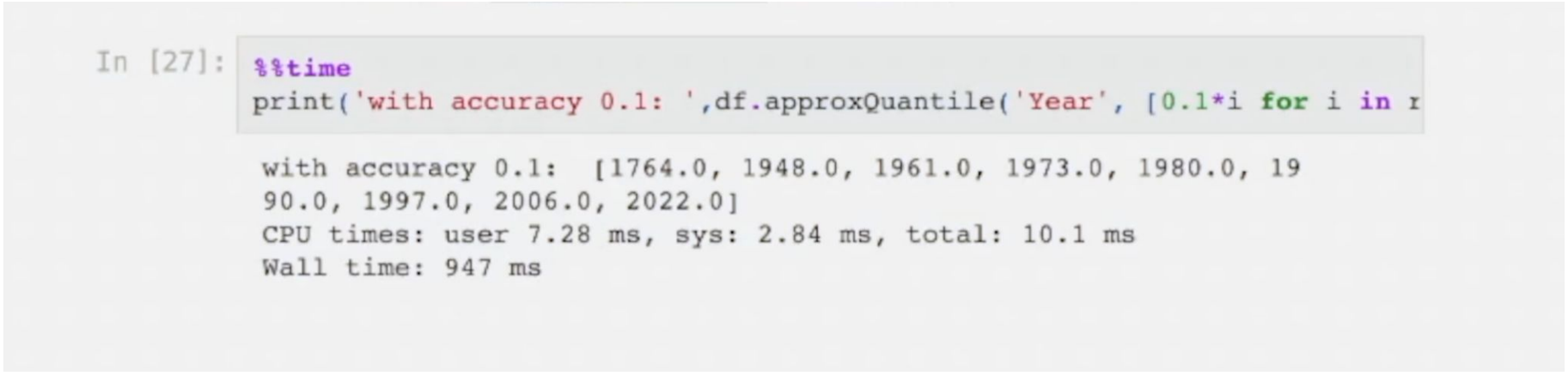
```
+------------------------------+
|approx_count_distinct(station)|
+------------------------------+
|                        128546|
+------------------------------+
```

Approximate Quantile

- Suppose we want to partition the years into 10 ranges
- Such that in each range we have approximately the same number of records.
- The method `approxQuantile` will use a sample to do this for us

Approximate Quantile

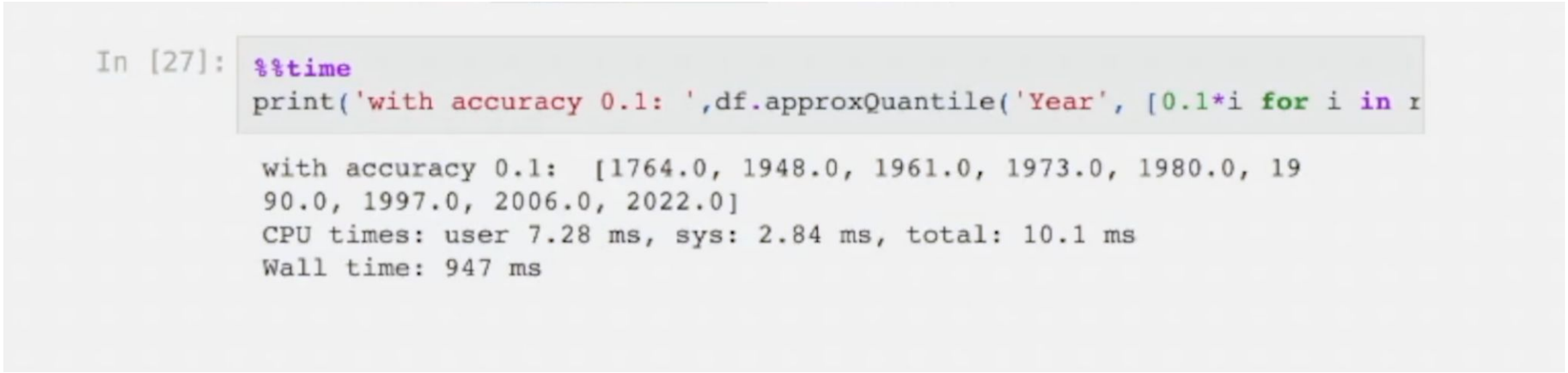
- Suppose we want to partition the years into 10 ranges
- Such that in each range we have approximately the same number of records.
- The method `approxQuantile` will use a sample to do this for us



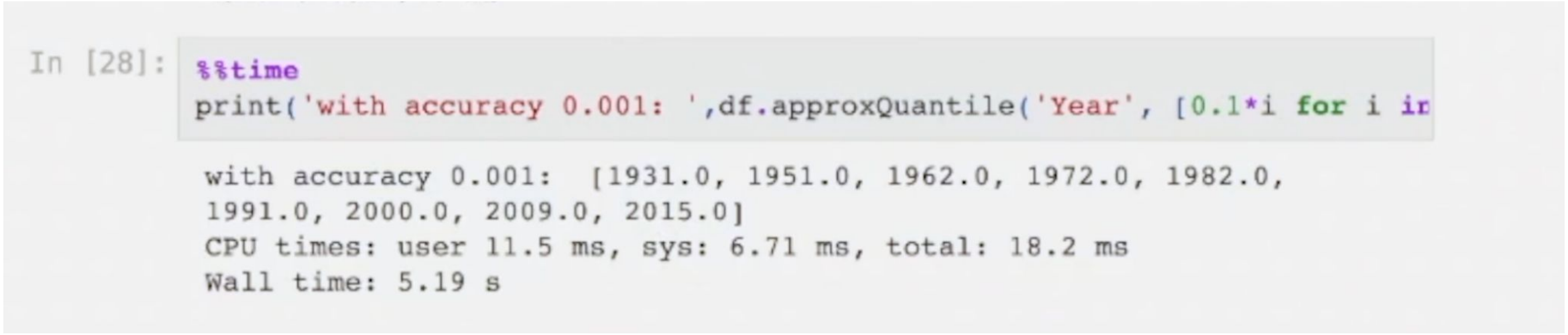
```
In [27]: %%time
         print('with accuracy 0.1: ',df.approxQuantile('Year', [0.1*i for i in r

         with accuracy 0.1:  [1764.0, 1948.0, 1961.0, 1973.0, 1980.0, 19
         90.0, 1997.0, 2006.0, 2022.0]
         CPU times: user 7.28 ms, sys: 2.84 ms, total: 10.1 ms
         Wall time: 947 ms
```

Approximate Quantile

- Suppose we want to partition the years into 10 ranges
- Such that in each range we have approximately the same number of records.
- The method `approxQuantile` will use a sample to do this for us



```
In [27]: %%time
         print('with accuracy 0.1: ',df.approxQuantile('Year', [0.1*i for i in r

         with accuracy 0.1:  [1764.0, 1948.0, 1961.0, 1973.0, 1980.0, 19
         90.0, 1997.0, 2006.0, 2022.0]
         CPU times: user 7.28 ms, sys: 2.84 ms, total: 10.1 ms
         Wall time: 947 ms
```

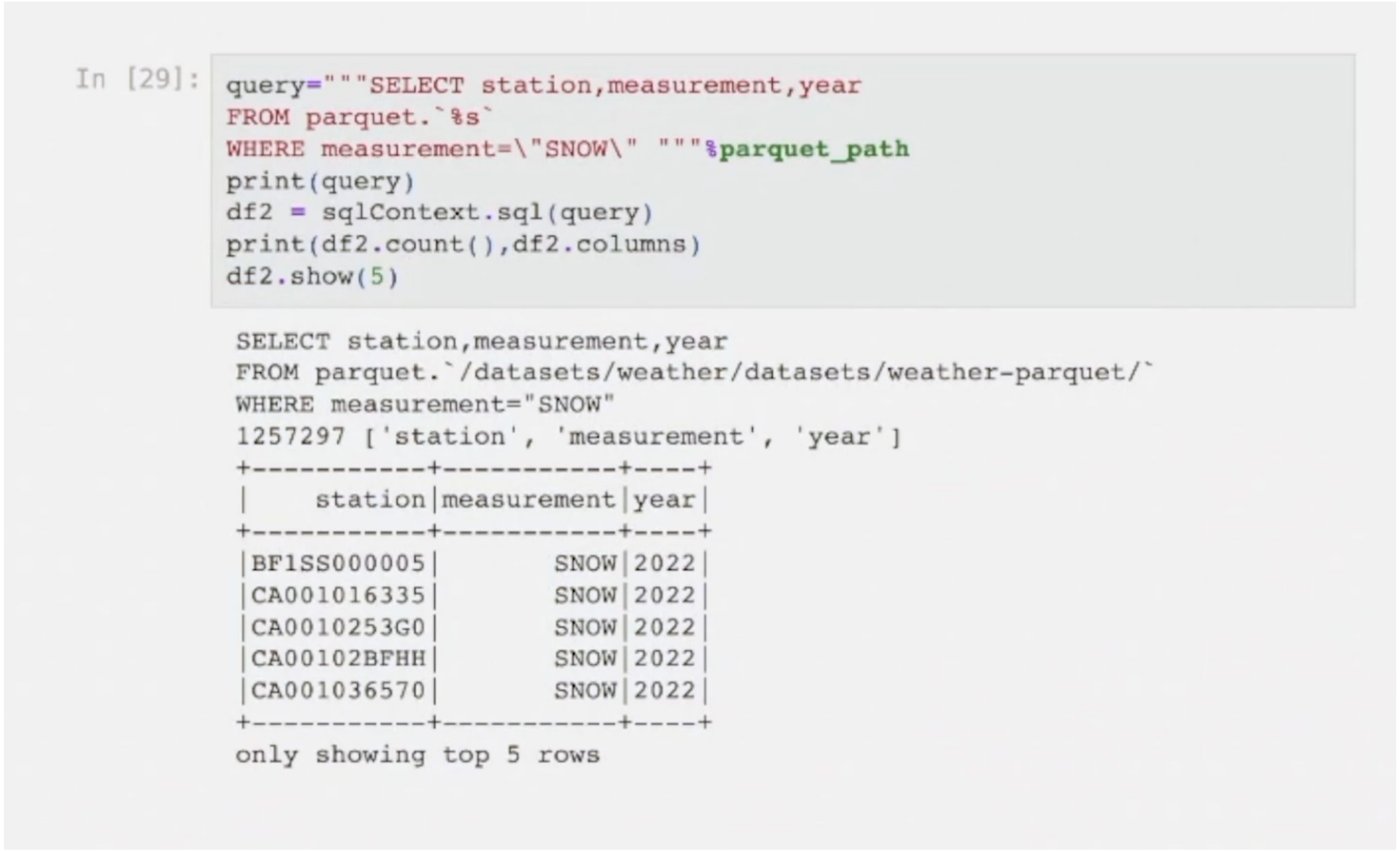


```
In [28]: %%time
         print('with accuracy 0.001: ',df.approxQuantile('Year', [0.1*i for i ir

         with accuracy 0.001:  [1931.0, 1951.0, 1962.0, 1972.0, 1982.0,
         1991.0, 2000.0, 2009.0, 2015.0]
         CPU times: user 11.5 ms, sys: 6.71 ms, total: 18.2 ms
         Wall time: 5.19 s
```

Reading rows selectively from Parquet

Suppose we are only interested in snow measurements. We can apply an SQL query directly to the Parquet files. As the data is organized in columnar structures, we can do the selection efficiently without loading the whole file to memory.

Here the file is small, but in real applications it can consist of hundreds of millions of records. In such cases loading the data first to memory and then filtering it is very wasteful.



```
In [29]: query="""SELECT station,measurement,year
         FROM parquet.`%s`
         WHERE measurement=\"SNOW\" """%parquet_path
         print(query)
         df2 = sqlContext.sql(query)
         print(df2.count(),df2.columns)
         df2.show(5)
```

```
SELECT station,measurement,year
FROM parquet.`/datasets/weather/datasets/weather-parquet/`
WHERE measurement="SNOW"
1257297 ['station', 'measurement', 'year']
+-----------+-----------+----+
|    station|measurement|year|
+-----------+-----------+----+
|BF1SS000005|       SNOW|2022|
|CA001016335|       SNOW|2022|
|CA0010253G0|       SNOW|2022|
|CA00102BFHH|       SNOW|2022|
|CA001036570|       SNOW|2022|
+-----------+-----------+----+
only showing top 5 rows
```

Summary

- Dataframes can be manipulated declaratively, which allows for more optimization.
- Dataframes can be stored and retrieved from Parquet files.
- It is possible refer to directly to a parquet file in an SQL query.
- See you next time!