Spark SQL

Module 3 | Chapter 2 | Notebook 2

In this lesson, we'll use hard drive data as the basis for our first big-data data set. `pyspark` enables you to access the Spark infrastructure and learn how and why you should use the `pyspark.sql.DataFrame` data type for large data sets.

Scenario: You are an employee in a large data center and are tasked with investigating server hard drive failures. Thanks to the monitoring team's excellent work, you have the error data of the last quarter for all the hard drives that your data center has in operation - roughly 30000.

Your boss has two questions for you:

1. Which hard drive models are responsible for the failures?
2. Which hard driver manufacturer should the company order replacement hard disks from?

The monitoring team has stored the data in the *HDD_logs/* folder.

Now let's look at the contents of the folder to get an overview of the data. We'll need the `os` module and its `os.listdir()` function.

`listdir()` requires the folder path as a `str` and then returns a `list` with all the file names in the folder.

Store the path in the variable `data_dir`. Then import `os` and store the output from `os.listdir()` in the variable `file_list`. Then sort `file_list` with the `sorted()` function and store the result again in `file_list`. Also print the number of files in the folder.

In [23]:
```
import os
data_dir = 'HDD_logs/'
file_list = sorted(os.listdir(data_dir))
len(file_list)
```

Out[23]:
```
91
```

As you can see, there are `91` individual data sets, which were gathered between 1.1.2016 and 31.3.2016. Each day has its own log file in CSV format. It would be a lot of work to analyze all these files one by one. We should therefore combine them into one large `DataFrame`. With `pandas` we would import the files as DataFrames and merge them. We could proceed as follows:

```
import pandas as pd

# create first DataFrame
```

```
print(file_list[0])
df = pd.read_csv(data_dir + file_list[0])

# append all remaining files to DataFrame
for file in file_list[1:]:
    print(file)
    tmp_df = pd.read_csv(data_dir + file)
    df = df.append(tmp_df()
```

Important: To avoid overloading your memory and then not being able to continue working, we have only included a picture of the code and the error message at this point. Take a close look at it.

```
import pandas as pd

#create first DataFrame
print(file_list[0])
df = pd.read_csv(data_dir + file_list[0])

#append all remaining files to DataFrame
for file in file_list[1:]:
    print(file)
    tmp_df = pd.read_csv(data_dir + file)
    df = df.append(tmp_df)
```

```
2016-01-01.csv
2016-01-02.csv
2016-01-03.csv
2016-01-04.csv
2016-01-05.csv
2016-01-06.csv
2016-01-07.csv
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-3-3e63cb56058d> in <module>
      9     print(file)
     10     tmp_df = pd.read_csv(data_dir + file)
---> 11     df = df.append(tmp_df)

~/.virtualenvs/data_scientist/lib/python3.6/site-packages/pandas/core/frame.py in append(self, other, ignore_index, verify_integrity, sort)
   6199         return concat(to_concat, ignore_index=ignore_index,
   6200                       verify_integrity=verify_integrity,
-> 6201                       sort=sort)
   6202
   6203     def join(self, other, on=None, how='left', lsuffix='', rsuffix='',
```


```
import pandas as pd

#create first DataFrame
print(file_list[0])
df = pd.read_csv(data_dir + file_list[0])

#append all remaining files to DataFrame
for file in file_list[1:]:
    print(file)
    tmp_df = pd.read_csv(data_dir + file)
    df = df.append(tmp_df)
```

```
2016-01-01.csv
2016-01-02.csv
2016-01-03.csv
2016-01-04.csv
2016-01-05.csv
2016-01-06.csv
2016-01-07.csv
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-3-3e63cb56058d> in <module>
      9     print(file)
     10     tmp_df = pd.read_csv(data_dir + file)
---> 11     df = df.append(tmp_df)

~/.virtualenvs/data_scientist/lib/python3.6/site-packages/pandas/core/frame.py in append(self, other, ignore_index, verify_integrity, sort)
   6199         return concat(to_concat, ignore_index=ignore_index,
   6200                       verify_integrity=verify_integrity,
-> 6201                       sort=sort)
   6202
   6203     def join(self, other, on=None, how='left', lsuffix='', rsuffix='',
```

```
import pandas as pd

#create first DataFrame
print(file_list[0])
df = pd.read_csv(data_dir + file_list[0])

#append all remaining files to DataFrame
for file in file_list[1:]:
    print(file)
    tmp_df = pd.read_csv(data_dir + file)
    df = df.append(tmp_df)
```

```
2016-01-01.csv
2016-01-02.csv
2016-01-03.csv
2016-01-04.csv
2016-01-05.csv
2016-01-06.csv
2016-01-07.csv
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-3-3e63cb56058d> in <module>
      9     print(file)
     10     tmp_df = pd.read_csv(data_dir + file)
---> 11     df = df.append(tmp_df)

~/.virtualenvs/data_scientist/lib/python3.6/site-packages/pandas/core/frame.py in append(self, other, ignore_index, verify_integrity, sort)
   6199         return concat(to_concat, ignore_index=ignore_index,
   6200                       verify_integrity=verify_integrity,
-> 6201                       sort=sort)
   6202
   6203     def join(self, other, on=None, how='left', lsuffix='', rsuffix='',
```

A `MemoryError` has occurred! After a few iterations of the `for` loop, the `data frame` `df` became so large that your memory isn't big enough to hold all the data at once. So we won't get very far with `pandas` here, unless we increase our computer's memory.

Don't worry, you don't need to unscrew your PC and install a new RAM module now. There is a more elegant way to deal with large amounts of data. With the hard drive data set, you're getting an insight into the world of **Big Data** and will use `pyspark` to analyze large amounts of data.

Big Data in PySpark

There is no doubt that the term *big data* has undoubtedly been a buzz word for some years now. Big data analytics processes data that achieve high values at the three **V**s, i.e. data that is gathered and processed:

- in large ***V**olumes*
- in a wide ***V**ariety*
- and at high ***V**elocity*

In order to get around the limitations of a single computer, like the one you just experienced, the data is stored and processed in a big-data context by a group of several computers. This combination is called a computing cluster, or just a cluster.

The good thing about a cluster is that no single computer has to do all the work. Instead, the task is broken down into parts and cleverly distributed to the computers in the cluster. A very common framework that makes this possible is **Apache Spark** or **Spark** for short. In the following notebooks we will use the `pyspark` API to send jobs to the Spark infrastructure using Python and process the results in Python.

If you want to learn more about big data and the Spark architecture, we recommend the following papers:

- Big Data
- Spark

PySpark

Pyspark can access a variety of modules from the Spark architecture. Here is an overview:


Spark Architecture

In this chapter, we will use Spark SQL. Later in the chapter you will also get to know Spark ML.

To access the Spark infrastructure, you must first install Spark and Java. You can then set up a cluster. In your company you should contact the data engineering department if you want to use `pyspark`. In the DataLab, we've already taken care of this for you. So you can use get started straight away.

In the first step you have to establish a connection to the **Spark SQL Driver**. For this you need `SparkSession` from the `pyspark.sql` module. The `SparkSession` object contains all the methods you need to generate a **SparkApp**. This allows Python to communicate with the Java and Scala code that Spark is based on. Now import `SparkSession`.

In [15]:
```
from pyspark.sql import SparkSession
```

Run the following code cell to establish a connection to the Spark SQL Driver. We use the round brackets to ignore the line breaks of Python and to give our code a nice structure.

It is possible you will see warning messages, for example `WARNING: An illegal reflective access operation has occurred`. You can ignore these warnings.

In [16]:
```
# connect to Spark
spark = (SparkSession
        .builder
        .appName("Python Spark SQL HDD Analysis")
        .getOrCreate()
        )
```

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/usr/local/spark-3.2.0-bin-hadoop3.2/jars/spark-unsafe_2.12-3.2.0.jar) to constructor java.nio.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsafe.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/04/30 21:15:01 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
```

`SparkSession.builder` contains the object that can create the connection. We assign a name to the connection with `SparkSession.builder.appName()`. Although this is only relevant when you aren't working in a Jupyter notebook, it's best practice to assign an app name.

`SparkSession.builder.appName().getOrCreate()` checks whether there is already a connection. If there is, this one is used, otherwise a new connection is created. The generated `SparkSession` variable `spark` now allows us to import the data.

Data analysis with Spark SQL is based on the ' `pyspark.sql.DataFrame`, which you can use almost exactly like a `DataFrame` from `pandas`. First let's try to create a `DataFrame` from one of the CSV files. The corresponding command is `my_SparkSession.read.csv()` and it expects a file path parameter. Additionally you should set the parameter `header=True` so that `pyspark` extracts the column names from the first row of the CSV file.

Create the DataFrame `df1` from the first file name in `file_list`. Then print the data types.

In [40]:
```
df1 = spark.read.csv(data_dir + file_list[0] , header=True);
df1.count()
```

Out[40]:
```
30597
```

`df1` has the file type `pyspark.sql.dataframe.DataFrame`.

To practice, create a second DataFrame `df2` from the second file in `file_list`. Then print the number of rows in `df2` using the `my_spark_df.count()` method.

In [31]:
```
df2 = spark.read.csv(data_dir + file_list[1] , header=True);
df2.count()
```

Out[31]:
```
30595
```

To join two DataFrames together, you need the `my_spark_df.union()` method. It takes a `pyspark.sql.DataFrame` as a parameter, which is then added to the end of the existing `pyspark.sql.DataFrame`. `my_spark_df.union()` is comparable to the `my_pandas_df.append()` method of a `pandas.DataFrame`.

Try it out now. Join `df1` and `df2` together to make `df1_2`. Then print out the number of rows again.

In [34]:
```
df1_2 = df1.union(df2)
df1_2.count()
```

Out[34]:
```
61192
```

The number of rows has almost doubled. So you successfully merged the two DataFrames. However, we would have still been able to process this many rows easily with `pandas`. Now let's merge all the files from `data_dir` into one `DataFrame`.

Create a `for` loop to do this. It should iterate through all the files in `file_list` and merges them into a variable `df`. It will take up to 2 minutes to run the loop. So print the current file name that is being processed at each iteration. This way you can see if the code is still running. At the end of the loop, print the number of rows in `df`.

Tip: Start by importing one file as a `DataFrame` outside the loop, and then add the remaining files to it.

In [41]:
```
df1 = spark.read.csv(data_dir + file_list[0] , header=True);

for file in file_list[1:]:
    df_temp = spark.read.csv(data_dir + file, header=True);
    df1 = df1.union(df_temp)

df1.count()
```

Out[41]:
```
2901729
```

Now all the data is contained in a single `DataFrame`. We won't need our `SparkSession` `spark` anymore in this lesson. It's good style to end the `SparkSession`:

In [42]:
```
spark.stop()
```

Congratulations: You have merged a lot of files into one `DataFrame` with `pyspark`. You have successfully solved the RAM problem by using a `pyspark.sql.DataFrame` which contains almost three million entries. In the next chapter we will look at more functionalities that the `pyspark.sql.DataFrame` has to offer and how you can use them for data analysis.

Remember:

- Big Data is *big* in the three **V**s : **V**olume, **V**ariety, and **V**elocity.
- `pyspark` gives you access to Spark, a framework to work with Big Data on computer clusters.
- If you want your `DataFrame` to contain more entries than your RAM can handle, use `pyspark.sql.DataFrame` from the `pyspark.sql` module.
- To use a `pyspark.sql.DataFrame`, you have to instantiate a `SparkSession` first:

```
# connect to Spark
  spark = (SparkSession
           .builder
           .appName("Python Spark SQL HDD Analysis")
           .getOrCreate()
           )
```

- It is best practice to close the `SparkSession` with `spark.stop()` once you have finished your work.

Literature: If you would like to delve deeper into the subject matter of this chapter, we recommend the following source(s):

- Su, Xiaomeng. 2018 Introduction to Big Data. NTNU, 2018. [Cited: 04 07, 2020.]
- Salloum, S., Dautov, R., Chen, X. et al. 2016. Big data analytics on Apache Spark. International Journal of Data Science and Analytics. 2016, Vol. 1.

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.
