Comparing DataFrames: spark vs pandas

Module 3 | Chapter 2 | Notebook 3

In this notebook we'll focus on the special features of the `pyspark.sql.DataFrame` and the similarities to the `pandas.DataFrame` and SQL queries. After this lesson, you'll will have:

- Got to know the term *lazy evaluation*.
- Displayed a `pyspark.sql.DataFrame`
- Transformed a `pyspark.sql.DataFrame`

Evaluation

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.

In this chapter we will compare the different DataFrames in `pandas` and `spark`. Import `pandas` with its conventional alias `pd`. Then instantiate a `SparkSession` object and name it `spark`. `SparkSession` is part of `pyspark.sql`.

Ignore warning messages if there are any.

In [1]:
```
import pandas as pd
from pyspark.sql import SparkSession

# 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/loc
al/spark-3.2.0-bin-hadoop3.2/jars/spark-unsafe_2.12-3.2.0.jar) to constructor java.ni
o.DirectByteBuffer(long,int)
WARNING: Please consider reporting this to the maintainers of org.apache.spark.unsaf
e.Platform
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective a
ccess 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(new
Level).
24/04/30 21:46:02 WARN NativeCodeLoader: Unable to load native-hadoop library for you
r platform... using builtin-java classes where applicable
```

Import the log file *2016-01-01.csv* into a `pandas.DataFrame` named `df_pandas` and into a `pyspark.sql.DataFrame` named `df_spark`. Then call `df_spark`.

In [6]:
```
data_dir = 'HDD_logs/'
df_pandas  = pd.read_csv(data_dir + '2016-01-01.csv')
df_spark = spark.read.csv(data_dir + '2016-01-01.csv', header=True)
```

You'll see straight away that the `spark.pyspark.sql.DataFrame` doesn't display any data at all, but only the names and the file format of the respective columns. The reason for this is that the `pyspark.sql.DataFrame` `df_spark` doesn't yet generate return values. This only happens when we send a query to the `DataFrame`, i.e. we only get access to the data when we use it explicitly. This concept is called **lazy evaluation** and saves a lot of memory space. When interacting with DataFrames, Spark distinguishes between two categories: ***Transformations*** und ***Actions***.

- Transformations are all operations that are applied to the entire data set, such as `select()`, `groupBy()`, `describe()`, `filter()`, etc.
- *Actions* are all operations that output data, such as `count()`, `sum()`, `average()`, `collect()`, and all operations that display data to the user.

Here's a brief overview:

Transformations	**Description**
`my_spark_df.select()`	Select one or more columns (as `pyspark.sql.DataFrame`)
`my_spark_df.groupBy()`	Group the data according to a column
`my_spark_df.orderBy()`	Order the data according to a column
`my_spark_df.distinct()`	Remove the duplicate lines
`my_spark_df.describe()`	Summarize the `DataFrame`
`my_spark_df.filter()`	Select rows that fulfill a condition

Actions	**Description**
`my_spark_df.count()`	Return number of rows

Actions	Description
`my_spark_df.show()`	Print the first rows of the DataFrame
`my_spark_df.collect()`	Return all rows as a list of `Row` objects
`my_spark_df.take()`	Return the given number of rows as a list of `Row` objects
`my_spark_df.head()`	Return the first rows

For you, this means that you can string together as many *transformations* as you like without receiving an output. To output something, you always need an *action* at the end. You can find an overview of all the attributes and methods of the `pyspark.sql.DataFrame` in the documentation.

Congratulations: Now you know that Spark only becomes active when an *action* is requested, due to lazy evaluation.

Display the `pyspark.sql.DataFrame`

First we'll compare the `my_df.head()` method. Look at the output for `df_pandas`.

```
In [8]:
# pandas
df_pandas.head()
```

Out[8]:

	date	serial_number	model	capacity_bytes	failure	smart_1_raw	smart_3_raw	sm
0	2016-01-01	MJ0351YNG9Z0XA	Hitachi HDS5C3030ALA630	3000592982016	0	0	540	
1	2016-01-01	Z305B2QN	ST4000DM000	4000787030016	0	54551400	0	
2	2016-01-01	MJ0351YNG9Z7LA	Hitachi HDS5C3030ALA630	3000592982016	0	0	566	
3	2016-01-01	MJ0351YNGABYAA	Hitachi HDS5C3030ALA630	3000592982016	0	0	507	
4	2016-01-01	PL1321LAG34XWH	Hitachi HDS5C4040ALE630	4000787030016	0	0	514	

Now display the first 5 rows of `df_spark` using the `my_spark_df.head(5)` method. Unlike `pandas` you have to specify a number here, because the default value is 1.

```
In [9]:
# spark
df_spark.head(5)
```

Out[9]:
```
[Row(date='2016-01-01', serial_number='MJ0351YNG9Z0XA', model='Hitachi HDS5C3030ALA63
0', capacity_bytes='3000592982016', failure='0', smart_1_raw='0', smart_3_raw='540',
smart_4_raw='14', smart_5_raw='0', smart_7_raw='0', smart_9_raw='27929', smart_10_raw
='0', smart_12_raw='14', smart_192_raw='216.0', smart_193_raw='216.0', smart_194_raw
='20.0', smart_197_raw='0', smart_198_raw='0', smart_199_raw='0'),
 Row(date='2016-01-01', serial_number='Z305B2QN', model='ST4000DM000', capacity_bytes
='4000787030016', failure='0', smart_1_raw='54551400', smart_3_raw='0', smart_4_raw
='4', smart_5_raw='0', smart_7_raw='2590344', smart_9_raw='411', smart_10_raw='0', sm
art_12_raw='4', smart_192_raw='0.0', smart_193_raw='16090.0', smart_194_raw='30.0', s
mart_197_raw='0', smart_198_raw='0', smart_199_raw='0'),
 Row(date='2016-01-01', serial_number='MJ0351YNG9Z7LA', model='Hitachi HDS5C3030ALA63
0', capacity_bytes='3000592982016', failure='0', smart_1_raw='0', smart_3_raw='566',
smart_4_raw='19', smart_5_raw='0', smart_7_raw='0', smart_9_raw='27492', smart_10_raw
='0', smart_12_raw='18', smart_192_raw='213.0', smart_193_raw='213.0', smart_194_raw
='24.0', smart_197_raw='0', smart_198_raw='0', smart_199_raw='0'),
 Row(date='2016-01-01', serial_number='MJ0351YNGABYAA', model='Hitachi HDS5C3030ALA63
0', capacity_bytes='3000592982016', failure='0', smart_1_raw='0', smart_3_raw='507',
smart_4_raw='15', smart_5_raw='0', smart_7_raw='0', smart_9_raw='26640', smart_10_raw
='0', smart_12_raw='15', smart_192_raw='241.0', smart_193_raw='241.0', smart_194_raw
='31.0', smart_197_raw='0', smart_198_raw='0', smart_199_raw='0'),
 Row(date='2016-01-01', serial_number='PL1321LAG34XWH', model='Hitachi HDS5C4040ALE63
0', capacity_bytes='4000787030016', failure='0', smart_1_raw='0', smart_3_raw='514',
smart_4_raw='29', smart_5_raw='0', smart_7_raw='0', smart_9_raw='24589', smart_10_raw
='0', smart_12_raw='29', smart_192_raw='40.0', smart_193_raw='40.0', smart_194_raw='2
7.0', smart_197_raw='0', smart_198_raw='0', smart_199_raw='0')]
```

As you can see, the raw representation of `df_spark` takes some getting used to. You display the data more clearly with the *action* `my_spark_df.show()`. By default, `pyspark` displays the first 20 rows in the `DataFrame`. You can adjust this number with the `n` parameter.

Use `my_spark_df.show()` to display the first 5 rows of `df_spark`.

In [11]:
```
df_spark.show(n=5)
```

```
+----------+--------------+--------------------+--------------+-------+-----------+--
---------+----------+----------+----------+----------+-----------+------------+-
------------+------------+------------+------------+------------+-------------+
|      date| serial_number|               model|capacity_bytes|failure|smart_1_raw|sm
art_3_raw|smart_4_raw|smart_5_raw|smart_7_raw|smart_9_raw|smart_10_raw|smart_12_raw|s
mart_192_raw|smart_193_raw|smart_194_raw|smart_197_raw|smart_198_raw|smart_199_raw|
+----------+--------------+--------------------+--------------+-------+-----------+--
---------+----------+----------+----------+----------+-----------+------------+-
------------+------------+------------+------------+------------+-------------+
|2016-01-01|MJ0351YNG9Z0XA|Hitachi HDS5C3030...| 3000592982016|      0|          0|
540|         14|          0|          0|      27929|           0|          14|
216.0|        216.0|         20.0|            0|            0|            0|
|2016-01-01|      Z305B2QN|         ST4000DM000| 4000787030016|      0|   54551400|
0|          4|          0|    2590344|        411|           0|           4|
0.0|      16090.0|         30.0|            0|            0|            0|
|2016-01-01|MJ0351YNG9Z7LA|Hitachi HDS5C3030...| 3000592982016|      0|          0|
566|         19|          0|          0|      27492|           0|          18|
213.0|        213.0|         24.0|            0|            0|            0|
|2016-01-01|MJ0351YNGABYAA|Hitachi HDS5C3030...| 3000592982016|      0|          0|
507|         15|          0|          0|      26640|           0|          15|
241.0|        241.0|         31.0|            0|            0|            0|
|2016-01-01|PL1321LAG34XWH|Hitachi HDS5C4040...| 4000787030016|      0|          0|
514|         29|          0|          0|      24589|           0|          29|
40.0|         40.0|         27.0|            0|            0|            0|
+----------+--------------+--------------------+--------------+-------+-----------+--
---------+----------+----------+----------+----------+-----------+------------+-
------------+------------+------------+------------+------------+-------------+
only showing top 5 rows
```

The display is still very confusing. This is because we have so many columns in the `DataFrame` that they are wider than our output. Let's take a look at what these columns contain. As with `pandas` you get the column names with the `my_spark_df.columns` attribute. Output them and see.

In [12]:
```
df_spark.columns
```

Out[12]:
```
['date',
 'serial_number',
 'model',
 'capacity_bytes',
 'failure',
 'smart_1_raw',
 'smart_3_raw',
 'smart_4_raw',
 'smart_5_raw',
 'smart_7_raw',
 'smart_9_raw',
 'smart_10_raw',
 'smart_12_raw',
 'smart_192_raw',
 'smart_193_raw',
 'smart_194_raw',
 'smart_197_raw',
 'smart_198_raw',
 'smart_199_raw']
```

From the column names we can see that we have two different groups of information available about the hard disks. The first group is metadata, which we can use to identify each individual hard disk. The second group is the **"S.M.A.R.T" data** (*Self-Monitoring, Analysis, and Reporting Technology*), which represents various hard disk parameters.

To find out which disk models are unreliable, we only need the metadata and the `faillure` column at the moment.

So we can program a query that selects the following columns from the `DataFrame` :

- `'date'`
- `'serial_number'`
- `'model'`
- `'capacity_bytes'`
- `'failure'`

You select columns in a `pyspark.sql.DataFrame` like you can in `pandas` . You can simply enclose a `str` or a `list` with the column names in the square brackets in `my_spark_df[]` . However, there isn't a spark equivalent to `my_pandas_df.loc[]` and `my_pandas_df.iloc[]` . To display the result of this query, we have to chain our selection with the *action* `my_spark_df.show()` with the `pyspark.sql.DataFrame` .

First save the required column names as a `list` in a variable named `meta_cols` . Then select only the `meta_cols` from `df_spark` and save the result as `df_spark_meta` . Then display the first five rows of `df_spark_meta` .

In [17]:
```
meta_cols = ['date', 'serial_number', 'model', 'capacity_bytes', 'failure']
df_spark_meta = df_spark[meta_cols]
df_spark_meta.show(n=5)
```

```
+----------+--------------+--------------------+--------------+-------+
|      date| serial_number|               model|capacity_bytes|failure|
+----------+--------------+--------------------+--------------+-------+
|2016-01-01|MJ0351YNG9Z0XA|Hitachi HDS5C3030...| 3000592982016|      0|
|2016-01-01|      Z305B2QN|         ST4000DM000| 4000787030016|      0|
|2016-01-01|MJ0351YNG9Z7LA|Hitachi HDS5C3030...| 3000592982016|      0|
|2016-01-01|MJ0351YNGABYAA|Hitachi HDS5C3030...| 3000592982016|      0|
|2016-01-01|PL1321LAG34XWH|Hitachi HDS5C4040...| 4000787030016|      0|
+----------+--------------+--------------------+--------------+-------+
only showing top 5 rows
```

Now do the same for `df_pandas` and store your selection in `df_pandas_meta` .

In [19]:
```
df_pandas_meta = df_pandas.loc[:,meta_cols]
df_pandas_meta.head()
```

Out[19]:

	date	serial_number	model	capacity_bytes	failure
0	2016-01-01	MJ0351YNG9Z0XA	Hitachi HDS5C3030ALA630	3000592982016	0
1	2016-01-01	Z305B2QN	ST4000DM000	4000787030016	0
2	2016-01-01	MJ0351YNG9Z7LA	Hitachi HDS5C3030ALA630	3000592982016	0
3	2016-01-01	MJ0351YNGABYAA	Hitachi HDS5C3030ALA630	3000592982016	0
4	2016-01-01	PL1321LAG34XWH	Hitachi HDS5C4040ALE630	4000787030016	0

Congratulations: This display isn't quite as pretty in the notebook as the `pandas` one, but it looks much nicer!

Transforming a `pyspark.sql.DataFrame`

Now let's return to our data: As we have already seen, the name for the "S.M.A.R.T" data is a bit cryptic, because we only see the ID of the "S.M.A.R.T" attribute, but not which attribute it is. Fortunately, the "S.M.A.R.T" attribute IDs are mostly standardized.

We've provided a description of the attributes in the file *SMART_attributes.csv*. It's not in a subfolder.

Load the data into a `pandas.DataFrame` named `description_df` and specify the `ID` column as the index.

In [21]:
```
description_df = pd.read_csv('SMART_attributes.csv')
description_df
```

Out[21]:

	ID	Attribute	Description
0	1	read_error_rate	Frequency of errors during read operations.
1	2	throughput_performance	Overall performance of a device.
2	3	spin_up_time	Time required a spindle to spin up to operatio...
3	4	start_stop_count	Estimated remaining life, based on the number ...
4	5	reallocated_sectors_count	The number of the unused spare sectors. When e...
...	...	...	...
73	250	read_error_retry_rate	There is no reliable information available abo...
74	251	total_nand_read_count	No Description Available
75	252	unknown_252	No Description Available
76	254	free_fall_protection	No Description Available
77	255	unknown_255	No Description Available

78 rows × 3 columns

Now we just have to replace the ID numbers in the column names with the corresponding attribute. This will be useful for the machine learning part. Write a loop that iterates through each `column` of `df_spark.columns`. In each step it should check whether `column` contains the substring `'smart_'`. If that is the case, then the loop should do the following:

- extract the attribute ID, i.e. the number between `'smart_'` and `'_raw'`
- look up the extracted attribute ID in `description_df` and temporarily store the 'str' store the `str` stored in the `'Attribute'` column
- put `'smart_'` in front of the `str` it looked up, so that we still know which columns contain the "S.M.A.R.T." data
- store the new compound name in `renamed_cols`

If the column name does not contain `'smart_'`, the name can be stored directly in `renamed_cols`.

In [54]:
```
renamed_cols = []
for column in df_spark.columns:
    if column.find('smart_') != -1:
        ID = column.split('_')[1]
        attribute =''

        mask = description_df.loc[:,'ID'] == pd.to_numeric(ID)

        if not description_df.loc[mask,['Attribute']].empty:
            attribute = description_df.loc[mask,['Attribute']].iloc[0].values[0]
            renamed_cols.append('smart_'+attribute)

    else:
        renamed_cols.append(column)

renamed_cols
```

Out[54]:
```
['date',
 'serial_number',
 'model',
 'capacity_bytes',
 'failure',
 'smart_read_error_rate',
 'smart_spin_up_time',
 'smart_start_stop_count',
 'smart_reallocated_sectors_count',
 'smart_seek_error_rate',
 'smart_power_on_hours_count',
 'smart_spin_up_retries',
 'smart_power_cycle_count',
 'smart_power_off_retract_cycles',
 'smart_load_unload_cycles',
 'smart_temperature',
 'smart_current_pending_sectors',
 'smart_off_line_uncorrectable',
 'smart_udma_crc_error_rate']
```

`renamed_cols` should look like this:

```
['date',
 'serial_number',
 'model',
 'capacity_bytes',
 'failure',
 'smart_read_error_rate',
 'smart_spin_up_time',
 'smart_start_stop_count',
 'smart_reallocated_sectors_count',
 'smart_seek_error_rate',
 'smart_power_on_hours_count',
 'smart_spin_up_retries',
 'smart_power_cycle_count',
 'smart_power_off_retract_cycles',
 'smart_load_unload_cycles',
 'smart_temperature',
 'smart_current_pending_sectors',
 'smart_off_line_uncorrectable',
 'smart_udma_crc_error_rate']
```

To rename all the column names of a `pyspark.sql.DataFrame` you need the `my_spark_df.toDF()`. You can pass any number of *strings* to them, and `pyspark` will rename all the columns from left to right according to the *strings* you pass and it stops when there are no more arguments or columns left. If you passed `renamed_cols` directly, Spark would print an error, because you only passed one argument, which has the `list` data type and not `str`.

We can use what we call `*args` here to solve this problem. If you put a `*` in front of a list and pass it to a function, all the elements of the list are treated as arguments their respective data types, separated by commas. This saves you a lot of typing.

Use the `my_spark_df.toDF()` method and `renamed_cols` to rename all columns in `df_spark`. Store the returned value again under `df_spark`. Then display the first five values in the `'smart_read_error_rate'` und `'smart_spin_up_time'` columns.

In [61]:
```
df_spark = df_spark.toDF(*renamed_cols)
df_spark.columns

df_spark['smart_read_error_rate', 'smart_spin_up_time'].show(n=5)
```

```
+---------------------+------------------+
|smart_read_error_rate|smart_spin_up_time|
+---------------------+------------------+
|                    0|               540|
|             54551400|                 0|
|                    0|               566|
|                    0|               507|
|                    0|               514|
+---------------------+------------------+
only showing top 5 rows
```

Now close the `SparkSession`.

In [62]:
```
spark.stop()
```

Congratulations: You have successfully imported and displayed your data set with `pyspark` and changed the column names. As you have seen, there are a lot similarities between the DataFrames from `pandas` and Spark, but there are also some differences.

Remember:

- If you want to modify a `pyspark.sql.DataFrame`, all the *transformations* are only applied when you generate an output, e.g. with `my_spark_df.show()`.
- DataFrame methods often have the same names in `pandas` and Spark.
- If you want to pass a lot of arguments at once, you can store them in a `list` and then pass them as `*list` (often referred to as `'*args'` in documentation).
