# Chapter: Three
# Working with RDD and Dataframe

## RDD, DataSet and DataFrame

Before we move towards optimization, lets get to know few spark's internals.

### RDD

RDD, Resilient Distributed Datasets, is a fundamental data structure of Spark. It is an immutable distributed collection of objects. All operations in spark ultimately happen through RDD.

Know more about RDD: *https://databricks.com/glossary/what-is-rdd*

### DataFrame and DataSet

In Spark, a DataFrame is a distributed collection of data organized into named columns. Think of this is like a database table.

Starting in Spark 2.0, Dataset takes on two distinct APIs characteristics: a strongly-typed API and an untyped API. Conceptually, consider DataFrame as an alias for a collection of generic objects Dataset[Row], where a Row is a generic untyped JVM object. Dataset, by contrast, is a collection of strongly-typed JVM objects, dictated by a case class you define in Scala or a class in Java.

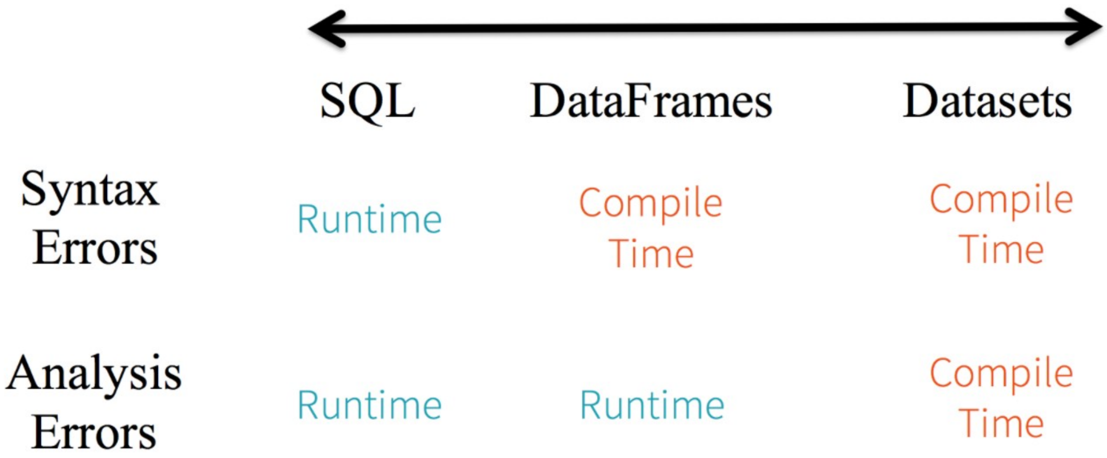
Is Datasets available with PySpark 2.0?

Short answerer is no, and log answerer is: As of Apache Spark 2.0.2, there is no native support for the Dataset API in Pyspark. There is support for Datasets only in Scala and Java. As Dataset is Strongly typed API and Python is dynamically typed means that runtime objects (values) have a type, as opposed to static typing where variables have a type. From Spark 2.0+ the Dataset API and Dataframe API are unified. Starting in Spark 2.0, Dataset takes on two distinct APIs characteristics: a strongly-typedAPI and an untyped API. Conceptually, consider DataFrame as an alias for a collection of generic objects

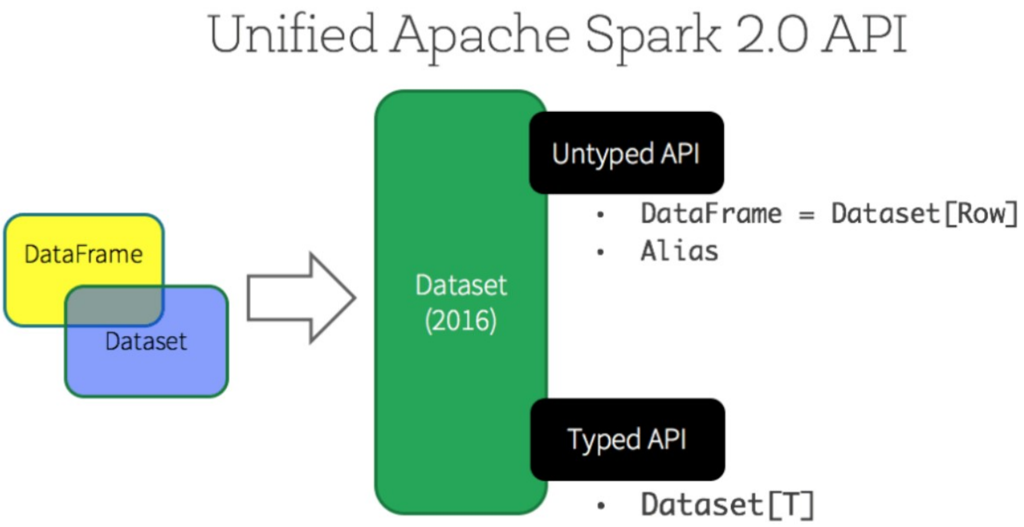
Dataset[Row], where a Row is a generic untypedJVM object. Dataset, by contrast, is a collection of strongly-typed JVM objects, dictated by a case class you define in Scala or a class in Java. So as Scala is strongly typed we have APIs as: Dataset[T] & DataFrame (alias for Dataset[Row]), in Python it is DataFram. From spark 2.0 the dataset and dataframe is unified.

| Language | Main Abstraction |
|---|---|
| Scala | Dataset[T] & DataFrame (alias for Dataset[Row]) |
| Java | Dataset[T] |
| Python | DataFrame |
| R | DataFrame |

***Note:*** *Since Python and R have no compile-time type-safety, we only have untyped APIs, namely DataFrames.*




*Effect of error in structured API*



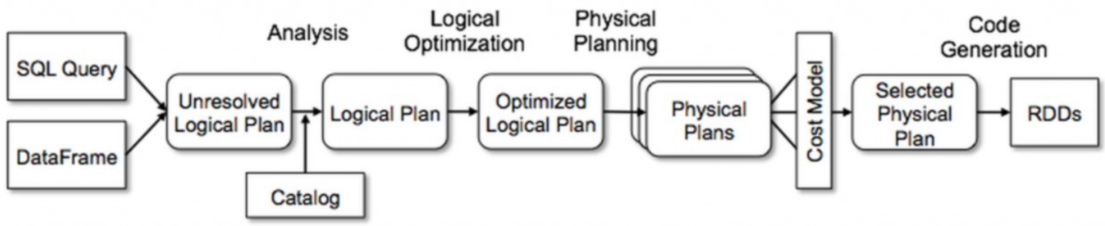

*Unified API in spark*

At the core of the Spark SQL engine are the catalyst optimizer and Tungsten. Together they support dataframe, dataset and spark sql APIs. These tools help spark to optimize it's workflow and perform better regardless of the language we use.

**Catalyst Optimizer**

This takes the user provided query and converts to an optimized execution plan and generates byte codes (RDD) for execution. It goes through four different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java byte code.

You can explore the execution plan by using ‘`df.explain(true)`’ command. We will use this pretty much every time we want to optimize the query.




Execution plan phases - from DataBricks

**Tungsten**

While generating Java byte code in the code generation phase, project Tungsten facilitates this operations and focuses on substantially improving the efficiency of memory and CPU for Spark applications, to push performance closer to the limits of modern hardware. Spark's shuffle subsystem, serialization and hashing (which are CPU bound) have been shown to be key bottlenecks, rather than raw network throughput of underlying hardware. The focus on CPU efficiency is motivated by the fact that Spark workloads are increasingly bottlenecked by CPU and memory use, rather than IO and network communication. Check the below link for more details.

*https://databricks.com/session/deep-dive-into-project-tungsten-bringing-spark-closer-to-bare-metal*

**Spark Encoder, Serialization and De-serialization**

Lets take a quick look into Saprk's Storage format. As spark is a memory intensive big data processing engine, utilizing efficient memory is crucial for spark success. Over the time spark's memory utilization has evolved and Project Tungsten played a major role in that. Bellow is the evolution of spark's storage format.

- Spark 1.0 to 1.3: It started with RDD's where data is represented as Java Objects.
- Spark 1.4 to 1.6: De-prioritized Java objects. DataSet and DataFrame evolved where data is stored in row-based format(Tungsten).
- Spark 2.x: Support for Vectorized Parquet which is columnar in-memory data is added.

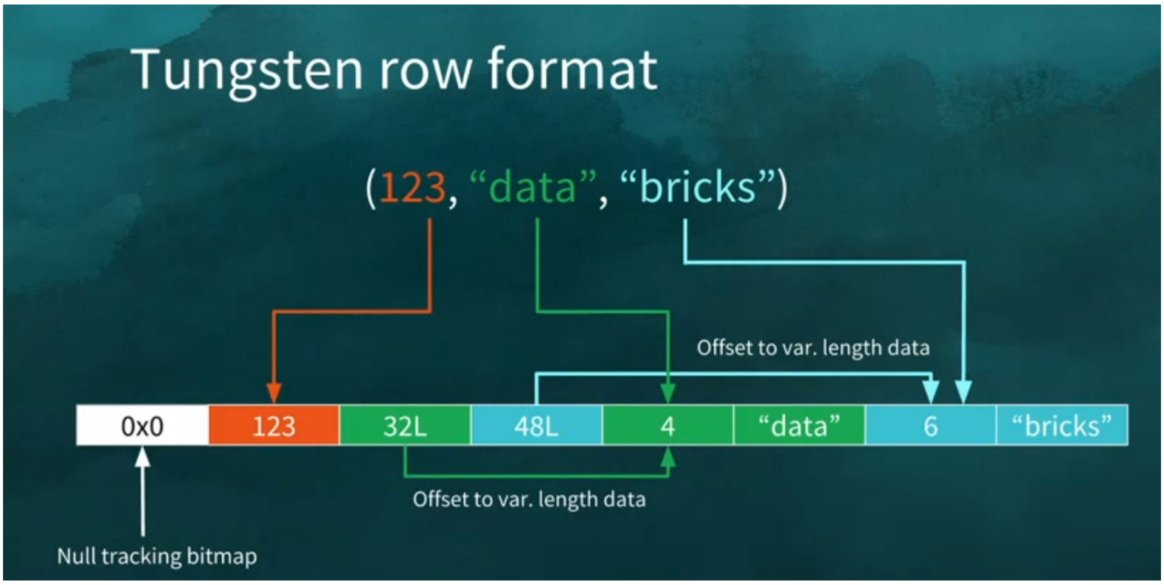
Representing data as a java object have limitations, java object have large overhead and java serialization, De-serialization and hashing is slow. During spark operation, transformation and shuffle, large amount of data travels between nodes in a cluster. serialization and deserialization is the process by which a typed object is encoded (serialized) into a binary presentation or format by the sender and decoded (deserialized) from binary format into its respective data-typed object by the receiver.

The JVM has its own built-in Java serializer and deserializer, but it's inefficient. To avoid the large overheads of a java object and slow java encoder/decoder, Spark adapted new binary row-based storage and a new faster spark encoder/decoder.

**How is an object stored in the new Binary-Row-Format?**

| | | |
|---|---|---|
| If the field is a primitive | With fixed length | Its stored in place |
| If the field is an Object | With variable length | 1. Its offset is stored in place.<br>2. At the specified offset, we store: *length of the variable + followed by its data* |

Following picture illustrates the same with an example. In this example, we took a tuple3 object (123, "data", "bricks") and and let's see how its stored in this new row-format.




*Example of tungsten row format*

- The first field 123 is stored in place as its primitive.
- The next 2 fields data and bricks are strings and are of variable length. So, an offset for these two strings is stored in place [32L and 48L respectively shown in the picture below].
- The data stored in these two offset's are of format "length + data". At offset 32L, we store 4 + data and likewise at offset 48L we store 6 + bricks.

**Note:** As tungsten keep primitive type data in row and variable length data as offset, to use the tungsten's memory effectively it's recommended to have the schema defined with proper data type. Do not keep all the data as string type.

**When to use RDD**

- Data is unstructured like text or media stream
- Do not care about the schema
- Wanted to have more control and flexibility by telling spark how to do rather than what to do
- Use of low level and type safe api
- Use of lambda functions

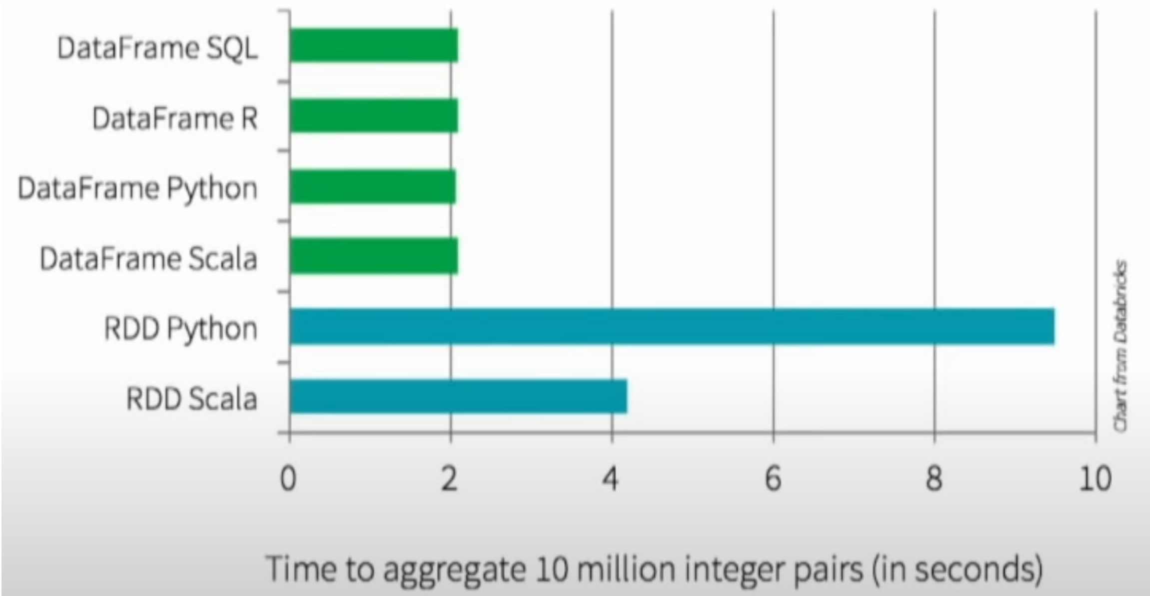
**Problem with RDD**

- RDD can not be optimized by spark optimizer due to opaque computations and data
- It has large memory foot prints
- Slow on non-JVM language like python
- Required more code to perform simple operation
- Developer might introduce inefficiencies in the code as we need to instruct spark on how to do

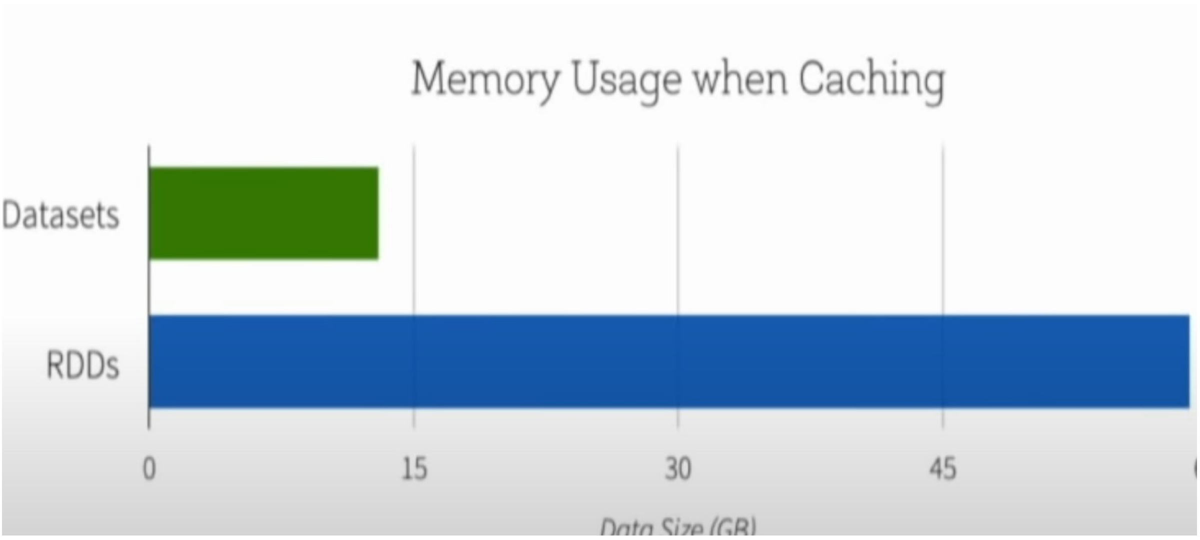
**When to use Dataset and DataFrame**

- Data is structured
- You care about schema

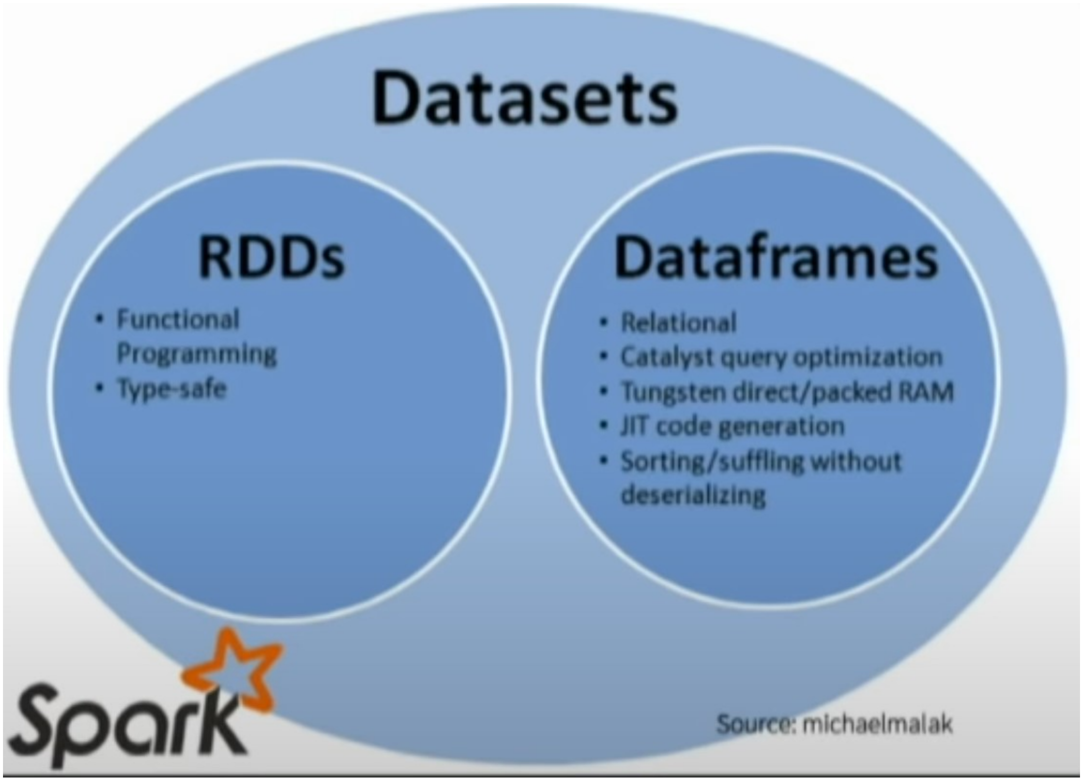
- Code optimization and better performance
- Care more about what to do than how to do
- Space efficiency and less memory foot prints
- Unified experiences and performance regardless of languages
- Required less code to perform complex operation




*Speed of computations - Dataframe vs RDD*



*Space efficiency - Dataset vs RDD*




*Best of both worlds - Dataset*

## Example of RDD vs DataFrame

**Example 1:** Reading CSV and apply filter

**RDD**

```
sales_data_rdd = sc.textFile('./data/*.csv')
header = sales_data_rdd.first() #extract header
sales_data_rdd_filter = sales_data_rdd \
                  .filter(lambda x: x != header) \
                  .map(lambda x: x.split(',')) \
                  .filter(lambda x: int(x[14]) >= 0) \
                  .filter(lambda x: int(x[5].split('/')[2]) > 2019) \
```

```
                        .filter(lambda x: x[1].lower().startswith('p')) \
                        .filter(lambda x: x[2].lower().find('office') !
= -1)
sales_data_rdd_filter.count()
```

Need 2 jobs, First one to read the header and then second job to run the count.

| Job Id ▾ | Description | Submitted | Duration | Stages: Succeeded/Total | Tasks (for all stages): Succeeded/Total |
|---|---|---|---|---|---|
| 1 | count at <stdin>:1<br>count at <stdin>:1 | 2021/04/20 19:04:00 | 37 s | 1/1 | 25/25 |
| 0 | runJob at PythonRDD.scala:166<br>runJob at PythonRDD.scala:166 | 2021/04/20 19:03:55 | 4 s | 1/1 | 1/1 |

Reading header took 4 sec and required one task, where COUNT spin up 25 tasks needed 37 sec, so total time is 41 sec.

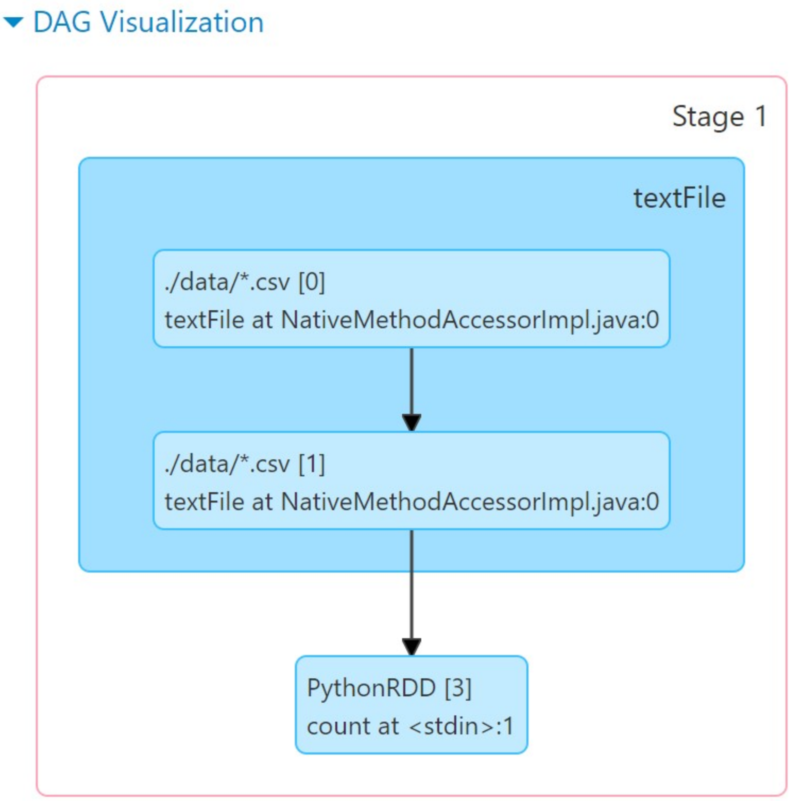
| Metric | Min | 25th percentile | Median | 75th percentile | Max |
|---|---|---|---|---|---|
| Task Deserialization Time | 9.0 ms | 41.0 ms | 1 s | 1 s | 1 s |
| Duration | 5 s | 5 s | 8 s | 8 s | 9 s |
| GC Time | 0.1 s | 0.2 s | 0.2 s | 0.2 s | 0.2 s |
| Result Serialization Time | 0.0 ms | 0.0 ms | 0.0 ms | 0.0 ms | 0.0 ms |
| Getting Result Time | 0.0 ms | 0.0 ms | 0.0 ms | 0.0 ms | 0.0 ms |
| Scheduler Delay | 23.0 ms | 32.0 ms | 71.0 ms | 74.0 ms | 79.0 ms |
| Peak Execution Memory | 0.0 B | 0.0 B | 0.0 B | 0.0 B | 0.0 B |
| Input Size / Records | 111.2 MiB / 890642 | 128.1 MiB / 1025174 | 128.1 MiB / 1025197 | 128.1 MiB / 1025238 | 128.1 MiB / 1033750 |

*Task details - input size*

On average each of 25 tasks read about 128 MB chunk of input files. 1st executor runs 8 tasks, 2nd one run 7 tasks and both 3rd and 4th executor run 5 tasks each. Also note the JVM heap and off Heap memory.

| Succeeded Tasks | Excluded | Input Size / Records | Peak JVM Memory OnHeap / OffHeap | Peak Execution Memory OnHeap / OffHeap | Peak Storage Memory OnHeap / OffHeap | Peak Pool Memory Direct / Mapped |
|---|---|---|---|---|---|---|
| 8 | false | 990.8 MiB / 7949695 | 21.6 MiB / 68.4 MiB | 0.0 B / 0.0 B | 539.8 KiB / 0.0 B | 1.1 MiB / 0.0 B |
| 7 | false | 879.6 MiB / 7050285 | 40.5 MiB / 68.1 MiB | 0.0 B / 0.0 B | 528.9 KiB / 0.0 B | 1 MiB / 0.0 B |
| 5 | false | 640.3 MiB / 5134579 | 26.3 MiB / 67 MiB | 0.0 B / 0.0 B | 528.9 KiB / 0.0 B | 1.1 MiB / 0.0 B |
| 5 | false | 606.6 MiB / 4865446 | 38.1 MiB / 67.7 MiB | 0.0 B / 0.0 B | 528.9 KiB / 0.0 B | 1 MiB / 0.0 B |

*Task details - jvm size*




*Execution plan of the RDD*

Points to note

- We did not define the schema. So it's very heard to work with the data. We were referring the columns by it's number.
- Spark had to read the entire file to remove the header.
- Spark had to read the entire file line by line and split the data by comma to get the individual columns, then we were able to work with the individual columns. So if our query required only one column, spark had to read the entire row, then split by comma to get to that column. This is very inefficient for a large dataset.
- We were not able to set the data types. Without the proper data type we had constantly type cast the data when required.
- We had to use lambda functions to work with the data. Lambda is anonymous and opaque to the Catalyst optimizer until runtime, when you use them it cannot efficiently discern what you're doing (you're

not telling Spark what to do) and thus cannot optimize your queries.

### Dataframe

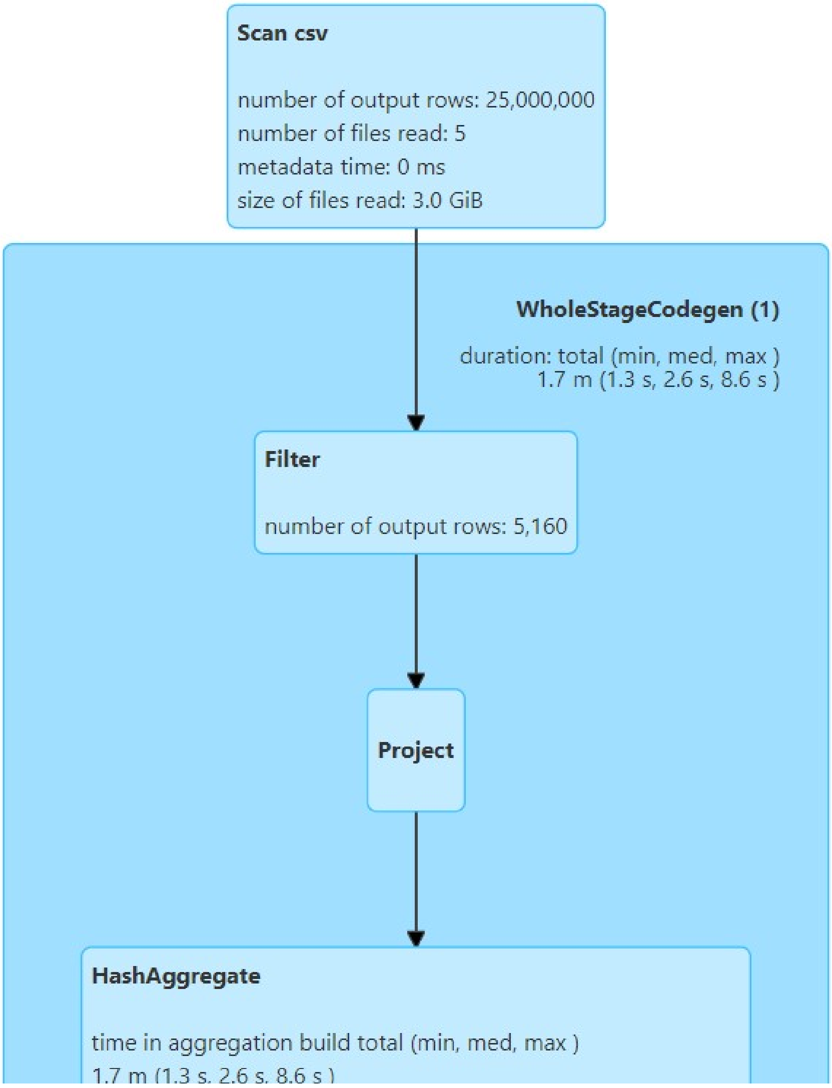
```
#pysaprk
spark.conf.set("spark.sql.files.maxPartitionBytes", 100000000)
schema = "`Region` STRING, \
          `Country` STRING, \
          `Item Type` STRING, \
          `Sales Channel` STRING, \
          `Order Priority` STRING, \
          `Order Date` STRING, \
          `Order ID` INT, \
          `Ship Date` STRING, \
          `Units Sold` DOUBLE, \
          `Unit Price` DOUBLE, \
          `Unit Cost` DOUBLE, \
          `Total Revenue` DOUBLE, \
          `Total Cost` DOUBLE, \
          `Total Profit` DOUBLE, \
          `id` INT"
# reading the csv with the schema
sales_data_df = spark.read.csv('./data/
*.csv',header=True, schema=schema)
from pyspark.sql.functions import *
sales_data_df_filter = sales_data_df.filter("id >= 0") \
                                  .filter(year(to_date('Order Date',
'M/d/yyyy')) > 2019) \
                                  .filter(lower('Country').startswith
('p')) \
                                  .filter(lower('Item Type').contains
('office'))
sales_data_df_filter.count()
```

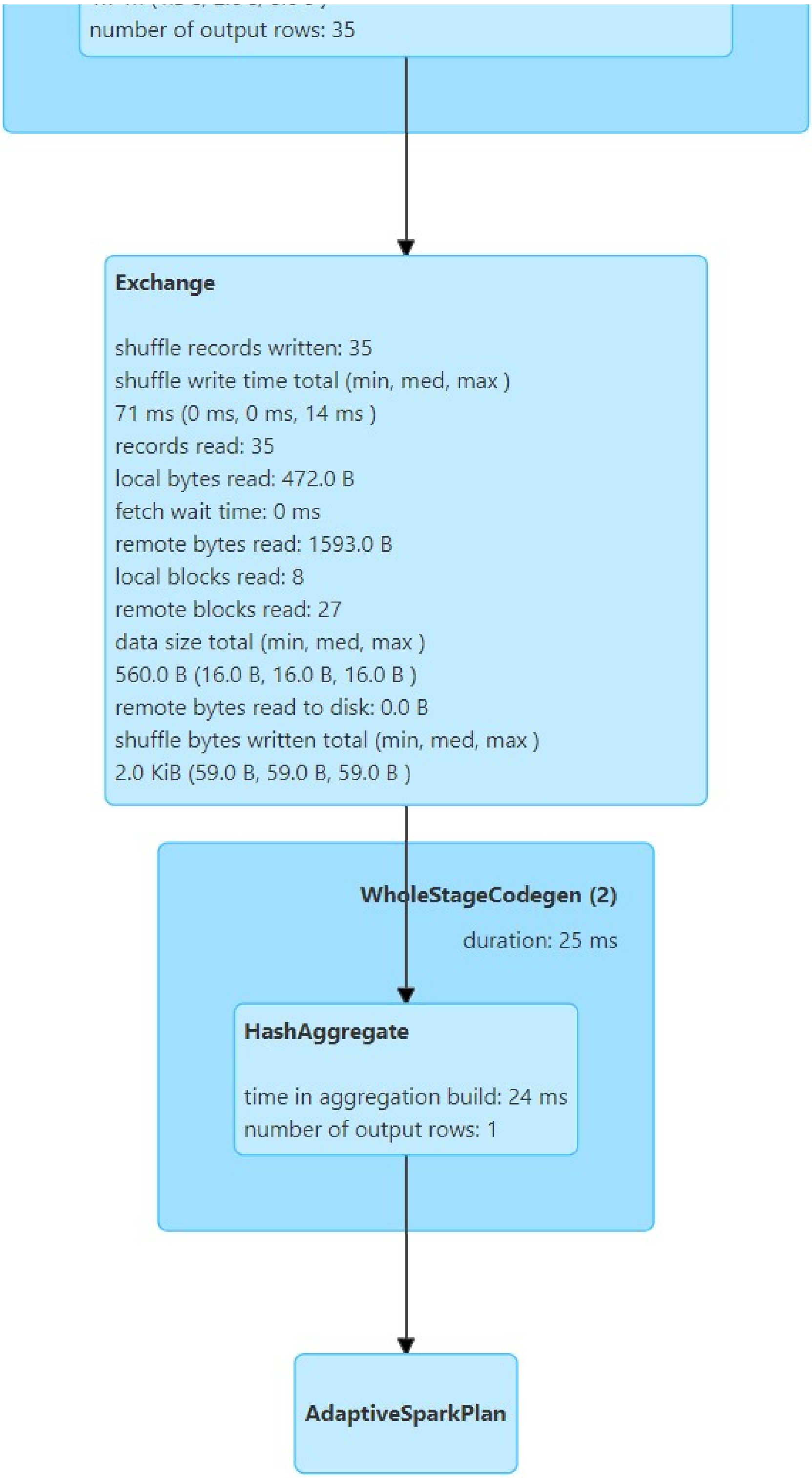
Spin up only one job and took 35 sec. It took 35 tasks to finish the count.

| Stage Id ▾ | Pool Name | Description | Submitted | Duration | Tasks: Succeeded/Total | Input | Output | Shuffle Read | Shuffle Write |
|---|---|---|---|---|---|---|---|---|---|
| 0 | default | count at NativeMethodAccessorImpl.java:0 +details | 2021/04/20 19:10:49 | 35 s | 35/35 | 3.0 GiB | | | 2.0 KiB |

*Number of tasks*

| Succeeded Tasks | Excluded | Input Size / Records | Shuffle Write Size / Records | Peak JVM Memory OnHeap / OffHeap | Peak Execution Memory OnHeap / OffHeap | Peak Storage Memory OnHeap / OffHeap | Peak Pool Memory Direct / Mapped |
|---|---|---|---|---|---|---|---|
| 11 | false | 960.9 MiB / 7706961 | 649 B / 11 | 39.9 MiB / 79.3 MiB | 0.0 B / 0.0 B | 573 KiB / 0.0 B | 1.1 MiB / 0.0 B |
| 10 | false | 909.9 MiB / 7287956 | 590 B / 10 | 39.1 MiB / 79.2 MiB | 0.0 B / 0.0 B | 573 KiB / 0.0 B | 1.1 MiB / 0.0 B |
| 8 | false | 719 MiB / 5770779 | 472 B / 8 | 24.4 MiB / 78.8 MiB | 0.0 B / 0.0 B | 573 KiB / 0.0 B | 1.1 MiB / 0.0 B |
| 6 | false | 528.1 MiB / 4234304 | 354 B / 6 | 31.6 MiB / 79 MiB | 0.0 B / 0.0 B | 573 KiB / 0.0 B | 1.1 MiB / 0.0 B |






*Execution plan of the dataframe*

Spark SQL also performs the same way.

**Spark sql**

```
# register the df as a temp table
sales_data_df.createOrReplaceTempView("sales_data_df")
spark.sql("""select * from sales_data_df where id >= 0 and year(to_da
te(`Order Date`, 'M/d/
yyyy')) > 2019 and lower(Country) like 'p%' and lower(`Item Type`) li
ke '%office%' """).count()
```

Points to note

- In data frame we had predefined the schema and data type, which help the spark Catalyst to optimize the query at run time.
- Working with the data is extremely easy as we are able to work with individual columns.

**<u>Example 2:</u>** Word count

This is one of the classic example for big data demonstration. I have a 15 GB of 'wiki_eng_articles' text file. I want to do words count and display top 10 most used words in the entire wiki English article. I ran this example using both RDD and dataframe, RDD ran littler slower than dataframe as expected.

**RDD**

```
# work count example

import time
import re

lines = sc.textFile('/mnt/e/Training_Data/wiki_eng_articles.txt')
words = lines.flatMap(lambda x: re.split(" +", x)).map(lambda w: (w,
1))
# count the words
counts = words.reduceByKey(lambda a, b: a + b)

# sort the data by counts - option 1
start_time = time.time()
countsSorted = counts.sortBy(lambda a: -a[1])
```

```
countsSorted.take(5)
print("--- %s seconds ---" % (time.time() - start_time))

# output
[('the', 175546618), ('of', 88955953), ('and', 73866826), ('in', 7122
1691), ('to', 49498126)]
--- 892.4747335910797 seconds ---

# sort the data by counts - option 2
start_time = time.time()
counts.takeOrdered(5, key = lambda x: -x[1])
print("--- %s seconds ---" % (time.time() - start_time))

# output
[('the', 175546618), ('of', 88955953), ('and', 73866826), ('in', 7122
1691), ('to', 49498126)]
--- 811.300785779953 seconds -–
```

### Dataframe

```
import time
from pyspark.sql import Row
from pyspark.sql.functions import explode, split, col

textFile = sc.textFile('/mnt/e/Training_Data/wiki_eng_articles.txt')

start_time = time.time()
df = textFile.map(lambda r: Row(r)).toDF(["line"])
df.select(explode(split(col("line"), "\s+")).alias("word")).groupBy("
word").count().orderBy("count", ascending=False).show(5,False)
print("--- %s seconds ---" % (time.time() - start_time))

# output
+----+---------+
|word|count    |
+----+---------+
|the |175546618|
|of  |88955953 |
|and |73866826 |
|in  |71221691 |
|to  |49498126 |
```

```
+----+---------+
--- 756.1691582202911 seconds ---
```

**<u>Example 3:</u>** Custom Parsing

This is, I would say, more complicated and closer to reality example.

Problem statement: I have 23 GB zipped text file with below format, which I want to parse and perform few data analysis.

This is a file from an on line book seller with all the details of the book it have in catalog.

Data Format:

```
/type/edition   /books/OL4686740M       2       2009-12-14T22:29:03.263605      {Publisher data in JSON f
```

*Input data format*

The data have 5 data blocks separated by 'spaces'. First data point is type of the data. It can be , second data point is the OnLine number of the book, third data point is the version number in integer format and fourth data point is in upload record date in date time format and last data block is in JSON format, having all the details of the book and publisher.

Approach:

1. Looking at the data format looks like data points are tab(\t) delimited.
2. We can use simple split function to split the data into columns.
3. Once we split the data we can use last element as the json string and can parse the json to get all the json elements.
4. First and second data points can be further separated using '/' delimiter.
5. We can create a python function which does all the data parsing process and return all parsed columns. In this way spark will loop the file once to get all the data parsed.
6. We can read the text file as RDD and call the python function as a map transformation to split the data.

7. We also need to implement data validated process. We can do this inside the python parse script. A simple validation would be to check for number of elements and check if the last element is a valid json.
8. Once we have the data parsed in a RDD, we can convert that to dataframe for further analysis.

Codes:

Saving the data:

I have saved the data in parquet format with snappy compression. I have also kept the output file size between 100mb to 200mb. In the output dir you will notice a '_success' file and 'snappy.parquet.crc' file.

'_success' - file indicates that the process is completed successfully. Some time I have noticed ETL designer uses this flag as spark job success flag and based on this they trigger subsequent jobs. This is not a great approach for a good ETL design. We should have these dependencies maintained inside a job orchestrator tool like Oozie or AirFlow.

'snappy.parquet.crc' - These files are the meta data files for the parquet output files, each output parquet file has one meta data file. One of the complain I noted that in amazon s3 writing these meta data files takes time and become bottleneck. You can choose to switch this off, however during my testing I have not noticed any performance improvement.

```
# switch off success file generation
spark.conf.set("mapreduce.fileoutputcommitter.marksuccessfuljobs",
"false")
# switch off the metadata
spark.conf.set("parquet.enable.summary-metadata", "false")
# enable direct output commiter
spark.conf.set("spark.hadoop.mapred.output.committer.class","com.apps
flyer.spark.DirectOutputCommitter")
```

Challenges:

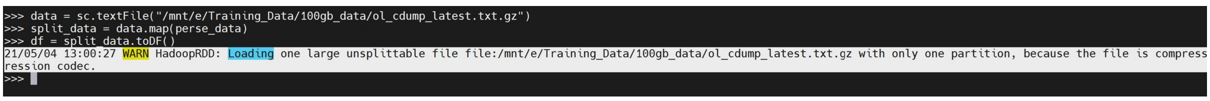
The input data is in zip format, which is not a splittable format. This means spark will use only one executor to read the file. This would be a slow process. One quick fix we can do is to unzip the file, but that will increase the file size. This is a good and quick option if we have enough space in the cluster. We can re-compress this file into splittable format, like bz2. However this will take considerable amount of time to uncompress and re-compress the file. The other option is to read the unzip file using spark, run all the clean process and then convert the data to more suitable file format like ORC+Snappy or perquet+Snappy for further analysis and process.

Here is the compression between different approaches.

| Data Format | Size | Preprocessing Time | Processing Time | Total |
|---|---|---|---|---|
| Gzip - single sile | 23 GB | 0 Min | 935 Min | 935 Min |
| Gzip - split into 42 gzip files | 23 GB | 40 Min | 365 Min | 405 Min |
| Uncompress file | 135 GB | 90 Min | 390 Min | 480 Min |

To parse the input data I have used python UDF. This approach is much slower and have negative performance impact. I will discuss this in the ne: section.

We do not know the schema details of the json column(last data point in the text file). To extract the schema from the data we have to read the entire data using 'spark.read.json' method. We can use 'samplingRation' to read fraction of the file, however this process is still going to take some time. If we know the schema then we can directly apply that on the data and save the processing time toe get schema.




*Can not split gzip file*

## Python UDF

User define function is very powerful way to extend spark functionality. You can build complex logic using UDF and apply that on data using sparks parallel processing. You can use UDF with RDD, Dataframe, dataset and spark SQL.

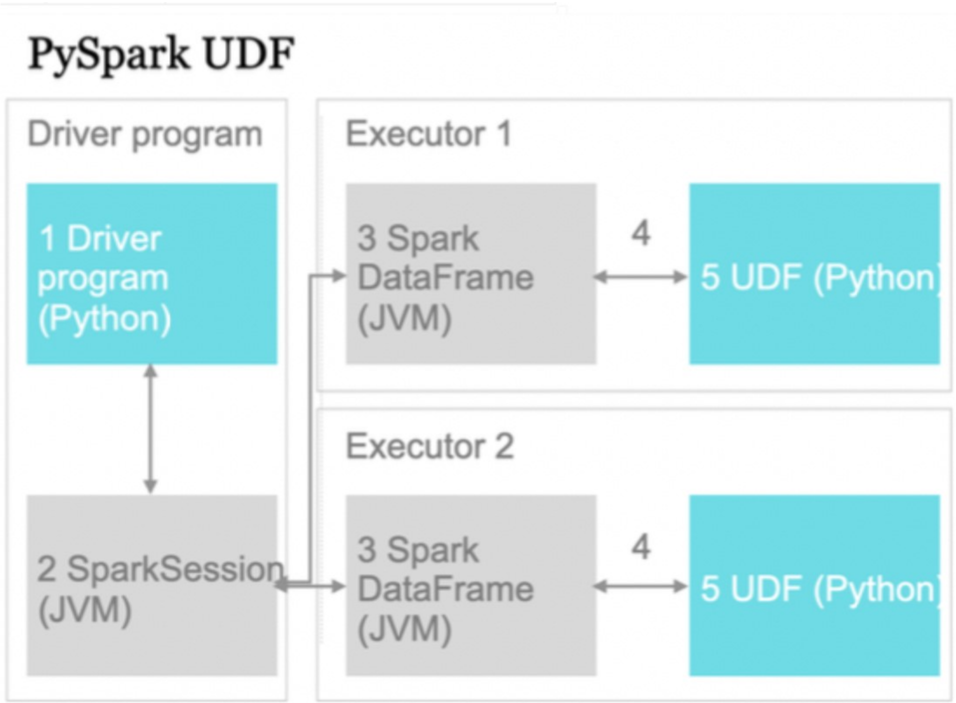
In previous example we have seen how to use python UDF with RDD. To use UDF with dataframe and spark SQL we need to register the UDF, once we define it.

Python UDF required the data to move between executor's JVM and python interpreter, which is very expansive.

We can speed up the process by using pandas UDF or vectorized UDF. This uses apache arrow, in memory columnar data format, to transfer data between JVM and python process. Once the data is in arrow format it can be consumed by both jvm and python and does not need to be serialize/de-serialize. This problem does not exist in scala UDF. We can careate a Python wrapper to call the Scala UDF from PySpark and found that we can bring the best of two worlds i.e. ease of Python coding and performance of Scala UDF. We have also found that creating a Python wrapper to call Scala UDF from PySpark code is around 15 times more performant than python UDFs.

One thing to aware is in PySpark/Spark does not guarantee the order of evaluation of subexpressions, meaning expressions are not guarantee to evaluate left-to-right or in any other fixed order. PySpark reorders the execution for query optimization and planning hence, AND, OR, WHERE and HAVING expression will have side effects. So when you are designing and using UDF, you have to be very careful especially with null handling as these results runtime exceptions. Always do NULL check, proper data type and error handling inside the UDF. For spark python udf is a black box and spark will not be able to do syntax check and performance optimization for the UDF. If the UDF fail then the entire spark process fail. Before you create any UDF, do your research to check if the similar function you wanted is already available in Spark SQL Functions.

Another point to note that if you are using any python packages which do not come per-install and needed to be installed using pip or anaconda install command, has to be installed on every spark worker node. If we do not have those packages install in every node then the spark process will fail on that node.




*Python UDF execution*

I have twitter data about covid in India, in JSON format. I need to clean the tweet text to apply NLP, to derive sentiment of the tweet. I want to remove HTML tag, spaces, new line charter, special characters, web url, email address, digit and date. I will create few python UDF and then apply those UDFs on twitter dataframe.

**Define the UDF**

```
# define UDF
# remove html tags from text
# Beautiful Soup ranks lxml's parser as being the best, then html5lib
's, then Python's built-in parser.
def remove_html(input):
# doing NULL check and proper datatype
```

```
    text = 'NULL' if input is None else str(input)
    # you can use html.parser like following code
    # soup = BeautifulSoup(text, "html.parser")
    soup = BeautifulSoup(text, "lxml")
    stripped_text = soup.get_text(separator=" ", strip=True)
    return stripped_text

 # conver the UDF and register in spark
remove_html_udf = udf(lambda row: remove_html(row), StringType())
spark.udf.register("remove_html_udf", remove_html, StringType())

#remove accented characters from text, e.g. café
def remove_accented_chars(text):
    text = 'NULL' if text is None else unidecode.unidecode(text)
    return text

# register the UDF
remove_accented_chars_udf = udf(lambda row: remove_accented_chars(row
), StringType())
spark.udf.register("remove_accented_chars_udf", remove_accented_chars
, StringType())
```

### Register the UDF

Now convert this function remove_html() to UDF by passing the function to PySpark SQL udf(). This function is available at org.apache.spark.sql.functions.udf package. Make sure you import this package before using it. PySpark SQL udf() function returns org.apache.spark.sql.expressions.UserDefinedFunction class object. The default type of the udf() is StringType.

```
# conver the UDF and register in spark
remove_html_udf = udf(lambda row: remove_html(row), StringType())
spark.udf.register("remove_html_udf", remove_html, StringType())
```

### Apply on Dataframe

```
pre_process_data_df = filter_data.withColumn("clean_tweet_text", remo
ve_accented_chars_udf(remove_html_udf("tweet_text")))
```

**Apply on Spark SQL**

Just call the UDF inside the spark sql statement. We can call multiple UDF in statement. Below is the example. You can get the entire code in github.

```
# apply the UDF in spark SQL way
Pre_process_data_sql = spark.sql("""select
                            id, lang, created_at, source,
                            user_id_str, user_name, user_location, us
er_description, tweet_text,
                            remove_accented_chars_udf(remove_html_udf
(tweet_text)) as clean_tweet_text
                            from filter_data""") \
                        .where("clean_tweet_text is not null")
```

**Note:**

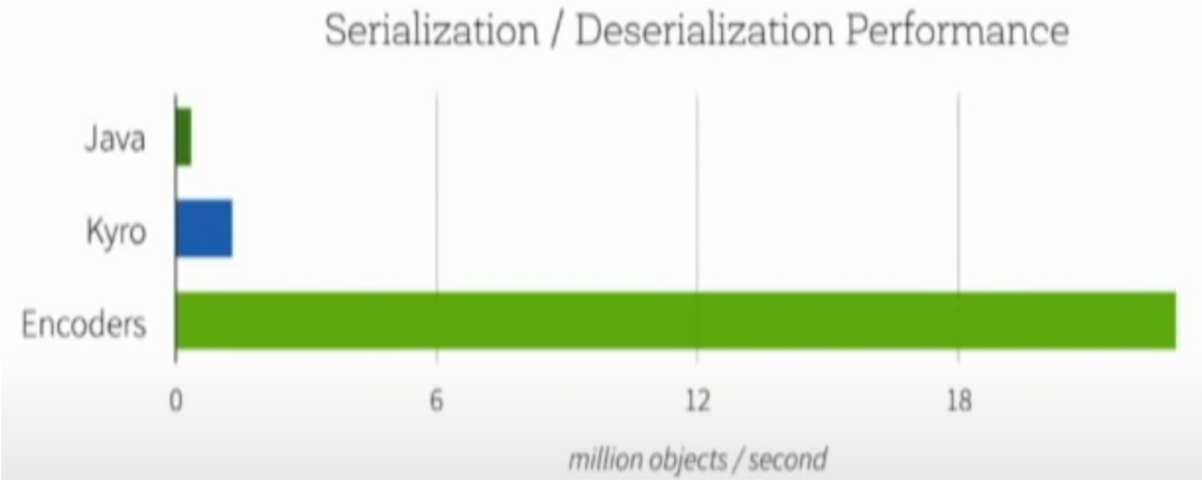
In the above example I had created two separate UDF, one to remove accented char and another to remove html tags from the input data. However, we can create a single UDF to achieve both functionality. Both approaches are fine based on the requirements. I have tested both approaches and did not find any performance gain one over another. However I like having multiple UDFs as this gives better flexibility, readability and easy to fix.

**Optimization Point**

- It is difficult to avoid using RDD while working with raw, unstructured text data. In this case, my suggestion would be, process the data using RDD and once it's become structured then switch to dataframe or dataset.
- Try to use Dataframe or dataset with pre-defined schema and proper data types.
- Avoid using lambda function with Data frame or dataset.
- Keep the input data split between 100MB and 200MB.
- Use splittable compressed format for the input data, if possible.
- If possible avoid using pure python udf, rather chose pandas udf or

scala udf wrapped in python.

- Use default spark encoder for better performances. Both java and Kryo encoder are slow, avoid using those.




*Encoder performance*