Title of Project: - Proactive Voltage Analysis using Pyspark

Tata power DDL

TATA POWER-DDL

Intern name: Ishita Sharma

IT – App Testing and Business Analytics

19/12/2022- 23/01/2023

Mentor: Mr Senthil Kumar

Contents

Section 1.0. What is Pyspark?

PySpark is a Spark library written in Python to run Python applications using Apache Spark capabilities, using PySpark we can run applications parallelly on the distributed cluster (multiple nodes).

Section 1.1. Introduction

PySpark is a Spark library written in Python to run Python applications using Apache Spark capabilities, using PySpark we can run applications parallely on the distributed cluster (multiple nodes).

The Spark Python API (PySpark) exposes the Spark programming model to Python. Apache Spark is an open source and is one of the most popular Big Data frameworks for scaling up your tasks in a cluster. It was developed to utilize distributed, in-memory data structures to improve data processing speeds.

In other words, PySpark is a Python API for Apache Spark. Apache Spark is an analytical processing engine for large scale powerful distributed data processing and machine learning applications.

Spark basically written in Scala and later on due to its industry adaptation its API PySpark released for Python using Py4J. Py4j is a Java library that is integrated within PySpark and allows python to dynamically interface with JVM objects, hence to run PySpark you also need Java to be installed along with Python, and Apache Spark.

PySpark is very well used in Data Science and Machine Learning community as there are many widely used data science libraries written in Python including NumPy, TensorFlow. Also used due to its efficient processing of large datasets. PySpark has been used by many organizations like Walmart, Trivago, Sanofi, Runtastic, and many more.

In real-time, PySpark has used a lot in the machine learning & Data scientist's community; thanks to vast python machine learning libraries. Spark runs operations on billions and trillions of data on distributed clusters 100 times faster than the traditional python applications.

Section 1.2. What is Proactive Voltage Analysis?

As the Smart Districts will have smart meters at consumer end. Therefore for pro-active observation of voltage profile of the consumers, we can utilize the smart meter data for this purpose.

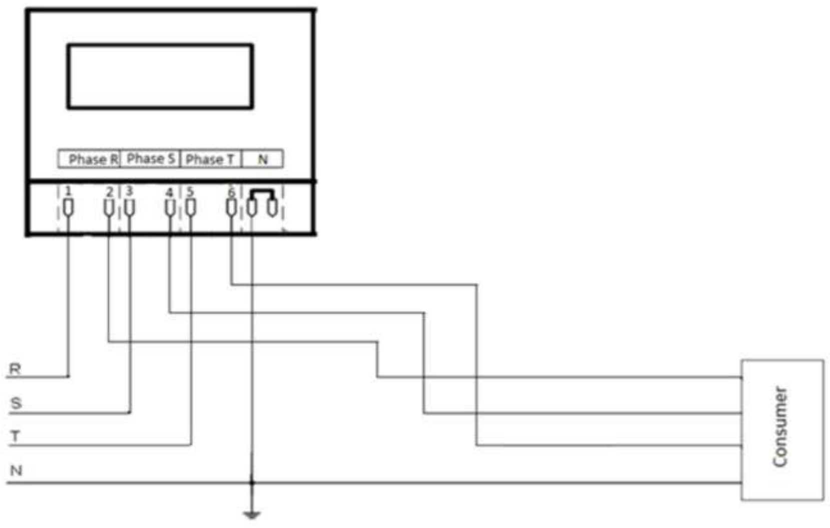
DTs in these smart districts will also have smart meters and with the analysis of data from DT meter, consumer meter along with grid 11kv bus voltage profile, we propose to develop one algorithm based UI which will predict the root cause of abnormal voltage faced by the consumer so that TATA Power-DDL can approach or solve the consumer voltage complaints before the consumer complains about this.

By proactively resolving conditions before they become serious issues, a utility can increase customer satisfaction and prevent equipment damage. On going monitoring of the system enables analysts to develop holistic solutions to voltage problems rather than simply react to the most recent customer complaint or problem. Following are just a few of the many advantageous outcomes voltage data provides:

- Identify high or low voltage trends on circuits
- Resolve voltage-related customer complaints
- Troubleshoot high or low voltage conditions
- Improve Volt/VAR optimization and conservation voltage reduction (CVR)
- Identify transformer winding failures
- Identify meter phase
- Identify and validate meter-to-transformer connectivity

Analysis of voltage data is key to achieving these benefits. With meter phase detection and transformer-to-meter connectivity analysis and validation, utilities can realize additional benefits such as improving reliability with more accurate outage monitoring as well as improving asset management programs.

Smart meters are providing data that has never been available beyond the substation. One of the best measures of the quality of a utility's product is voltage, which is measured at the point of sale for every utility customer. Smart meters communicate the information to the consumer for greater clarity of consumption behaviour, and electricity suppliers for system monitoring and customer billing. Smart meters typically record energy near real-time, and report regularly, short intervals throughout the day. Smart meters enable two-way communication between the meter and the central system.



Section 1.3. Importance of analysis of Smart Meter

Smart meter technology not only holds great potential in terms of achieving environmental and climate targets in an overall economic context, but also has benefits for individual end consumers (households and companies) and energy companies when deployed. The new metering systems should not only ensure secure transmission of metering data to the energy company, but also lead to increased transparency for the end customer about actual electricity consumption. In addition to providing better insight into and control over consumption data, the new applications and products can also help to reduce energy consumption and increase energy efficiency.

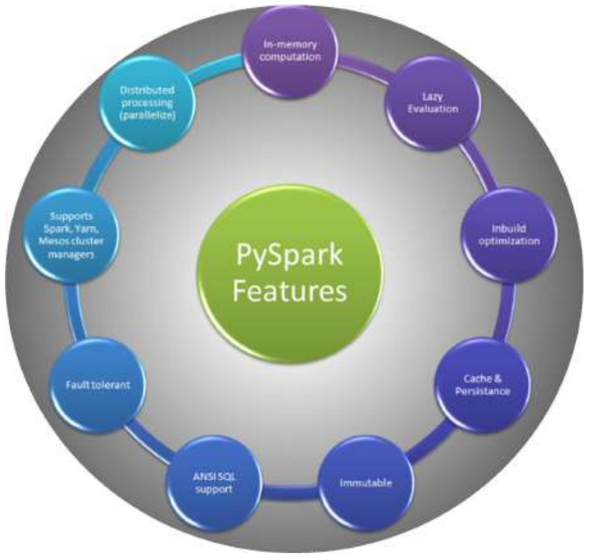
Furthermore, the measured data can be used to check whether the power grid is working optimally and the power held in reserve (in kilowatt) in the grid can be optimized by comparing it with real values. A reduction in the power held in reserve has a direct impact on electricity prices. Even expensive second or even minute reserves, which have to be produced by gas turbines to stabilize the grid, could contribute to low-cost supply security for all consumers through the remote controllability and short-term disconnection of consumers via a large number of smart meters. In addition, metering data could also be used to perform sizing of generation plants and storage (e.g., battery storage). This would solve the problem of missing design data for many engineering companies when planning, for example, energy concepts in practice.

Similar measurement values already exist for consumers of registered power measurement, these have a meter which transmits quarter-hourly power data in kilowatt to the responsible grid operator, thereby the power grid can be operated safely, since power peaks are identified early. With this data, which can be made available to companies in csv or parquet format on request, peak load management can be optimized or even new plants can be designed. This can also be made possible for households and small companies with smart meter technology. Following the three stages of analytics, namely, descriptive, predictive, and prescriptive analytics, we identify the critical application areas as load analysis, load forecasting, and load management.

Section 1.4. Features and Advantages of Pyspark

FEATURES:

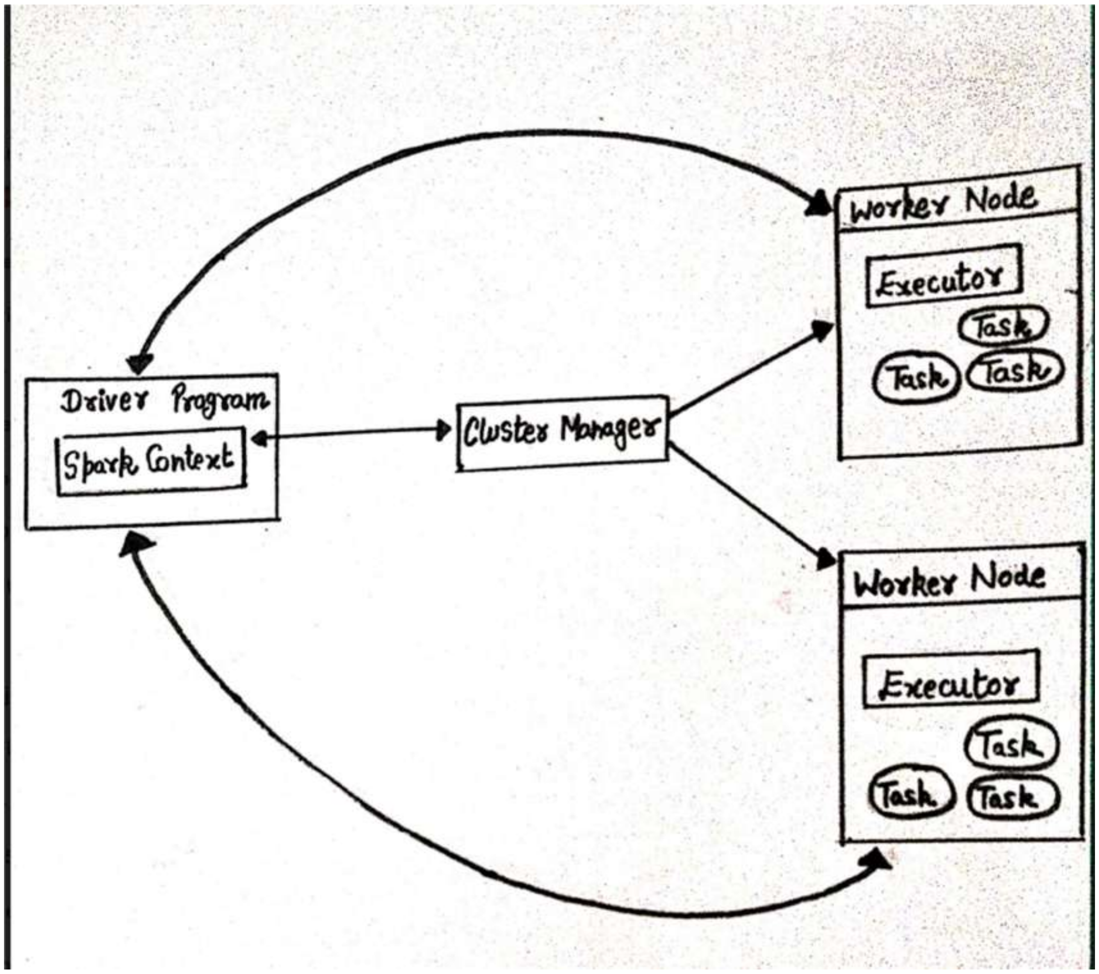
- In-memory computation
- Distributed processing using parallelize
- Can be used with many cluster managers (Spark, Yarn, Mesos etc)
- Fault-tolerant
- Immutable
- Lazy evaluation
- Cache & persistence
- Inbuilt-optimization when using Data Frames
- Supports ANSI SQL



ADVANTAGES:

- PySpark is a general-purpose, in-memory, distributed processing engine that allows you to process data efficiently in a distributed fashion.
- Applications running on PySpark are 100x faster than traditional systems.
- You will get great benefits using PySpark for data ingestion pipelines.
- Using PySpark we can process data from Hadoop HDFS, AWS S3, and many file systems.
- PySpark also is used to process real-time data using Streaming and Kafka.
- Using PySpark streaming you can also stream files from the file system and also stream from the socket.
- PySpark natively has machine learning and graph libraries.

Section 1.5. Spark architecture



When the Driver Program in the Apache Spark architecture executes, it calls the real program of an application and creates a SparkContext. SparkContext contains all of the basic functions. The Spark Driver includes several other components, including a DAG Scheduler, Task Scheduler, Backend Scheduler, and Block Manager, all of which are responsible for translating user-written code into jobs that are actually executed on the cluster.

The Cluster Manager manages the execution of various jobs in the cluster. Spark Driver works in conjunction with the Cluster Manager to control the execution of various other jobs. The cluster Manager does the task of allocating resources for the job. Once the job has been broken down into smaller jobs, which are then distributed to worker nodes, SparkDriver will control the execution.
Many worker nodes can be used to process an RDD created in the SparkContext, and the results can also be cached.

The Spark Context receives task information from the Cluster Manager and enqueues it on worker nodes. The executor is in charge of carrying out these duties. The lifespan of executors is the same as that of the Spark Application. We can increase the number of workers if we want to improve the performance of the system. In this way, we can divide jobs into more coherent parts.

Spark Architecture Applications:

The Spark driver

The master node (process) in a driver process coordinates workers and oversees the tasks. Spark is split into jobs and scheduled to be executed on executors in clusters. Spark contexts (gateways) are created by the driver to monitor the job working in a specific cluster and to connect to a Spark cluster. In the diagram, the driver programmes call the main application and create a spark context (acts as a gateway) that jointly monitors the job working in the cluster and connects to a Spark cluster. Everything is executed using the spark context.

Each Spark session has an entry in the Spark context. Spark drivers include more components to execute jobs in clusters, as well as cluster managers. Context acquires worker nodes to execute and store data as Spark clusters are connected to different types of cluster managers. When a process is executed in the cluster, the job is divided into stages with gain stages into scheduled tasks.

The Spark executors

An executor is responsible for executing a job and storing data in a cache at the outset. Executors first register with the driver programme at the beginning. These executors have a number of time slots to run the application concurrently. The executor runs the task when it has loaded data and they are removed in idle mode. The executor runs in the Java process when data is loaded and removed during the execution of the tasks. The executors are allocated dynamically and constantly added and removed during the execution of the tasks. A driver program monitors the executors during their performance. Users' tasks are executed in the Java process.

Cluster Manager

A driver program controls the execution of jobs and stores data in a cache. At the outset, executors register with the drivers. This executor has a number of time slots to run the application concurrently. Executors read and write external data in addition to servicing client requests. A job is executed when the executor has loaded data and they have been removed in the idle state. The executor is dynamically allocated, and it is constantly added and deleted depending on the duration of its use. A driver program monitors executors as they perform users' tasks. Code is executed in the Java process when an executor executes a user's task.

Worker Nodes

The slave nodes function as executors, processing tasks, and returning the results back to the spark context. The master node issues tasks to the Spark context and the worker nodes execute them. They make the process simpler by boosting the worker nodes (1 to n) to handle as many jobs as possible in parallel by dividing the job up into sub-jobs on multiple machines. A Spark worker monitors worker nodes to ensure that the computation is performed simply. Each worker node handles one Spark task. In Spark, a partition is a unit of work and is assigned to one executor for each one.

Modes of Execution:

You can choose from three different execution modes: local, shared, and dedicated. These determine where your app's resources are physically located when you run your app. You can decide where to store resources locally, in a shared location, or in a dedicated location.

1. Cluster mode
2. Client mode
3. Local mode

Cluster mode: Cluster mode is the most frequent way of running Spark Applications. In cluster mode, a user delivers a pre-compiled JAR, Python script, or R script to a cluster manager. Once the cluster manager receives the pre-compiled JAR, Python script, or R script, the driver process is launched on a worker node inside the cluster, in addition to the executor processes. This means that the cluster manager is in charge of all Spark application-related processes.

Client mode: In contrast to cluster mode, where the Spark driver remains on the client machine that submitted the application, the Spark driver is removed in client mode and is therefore responsible for maintaining the Spark driver process on the client machine. These machines, usually referred to as gateway machines or edge nodes, are maintained on the client machine.

Local mode: Local mode runs the entire Spark Application on a single machine, as opposed to the previous two modes, which parallelized the Spark Application through threads on that machine. As a result, the local mode uses threads instead of parallelized threads. This is a common way to experiment with Spark, try out your applications, or experiment iteratively without having to make any changes on Spark's end.

Section 1.6. Pyspark modules and packages

- Pyspark RDD
- Pyspark Data Frame and SQL
- Pyspark streaming
- Pyspark MLlib
- Pyspark graphs frames
- Pyspark resource

Section 1.7. Difference between Spark and Pandas data frames

Spark data frame	**Panda data frame**
Spark supports parallelization	Pandas does not support parallelization
It has a multiple nodes	Pandas has only single node
It follows Lazy evaluation which means that task is not performed until it is executed	In Pandas, Eager evaluation is performed that means the task is performed immediately
Spark Data frames are immutable	Pandas data frame are mutable
Complex operations are difficult to perform	Complex operations are easier to perform
Spark is distributed hence processing will be faster on large amount of data	Pandas aren't distributed hence processing will be slower
sparkDataFrame.count() returns the number of rows	pandaDataFrame.count() returns the null observations in the columns
Spark assures the fault tolerance	Pandas does not assures the fault tolerance
It is excellent for building a scalable application.	It can't be used to build a scalable application.

Section 2.0. Installing Pyspark

PySpark is included in the official releases of Spark available in the Apache Spark website. For Python users, PySpark also provides pip installation from PyPI. This is usually for local usage or as a client to connect to a cluster instead of setting up a cluster itself.

For installing PySpark by using pip, Conda, downloading manually, and building from the source.

Section 2.1. Pyspark – SparkSession

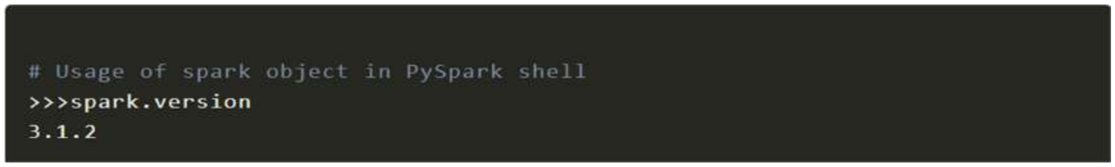
SparkSession is an entry point to underlying PySpark functionality in order to programmatically create PySpark RDD, Data Frame. Its object spark is default available in pyspark-shell and it can be created programmatically using SparkSession. It will be created using **SparkSession.builder()**.

Though SparkContext used to be an entry point prior to 2.0, it is not completely replaced with SparkSession, many features of SparkContext are still available and used in Spark 2.0 and later. You should also know that SparkSession internally creates SparkConfig and SparkContext with the configuration provided with SparkSession.

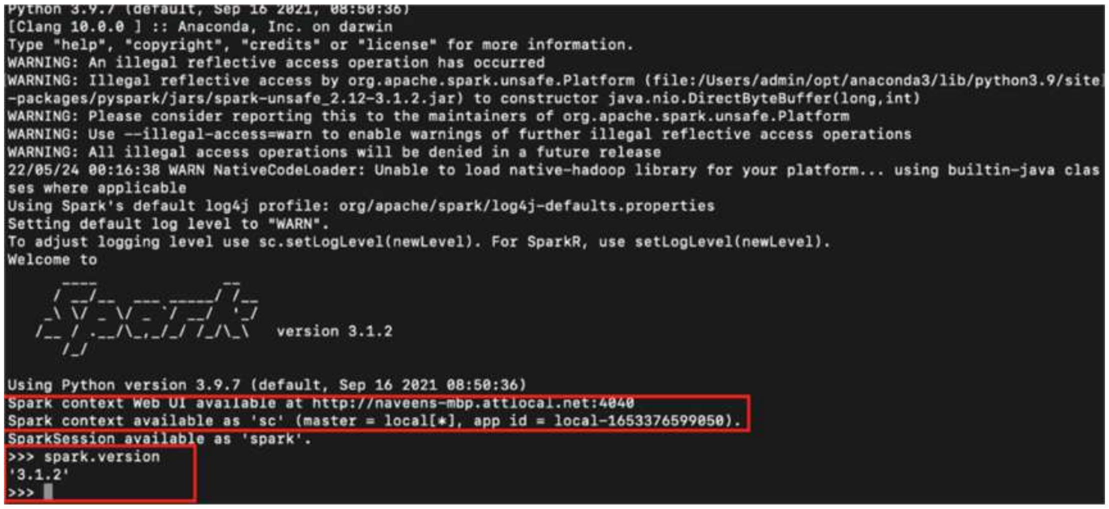
SparkSession also includes all the APIs available in different contexts –

- SparkContext,
- SQLContext,
- StreamingContext,
- HiveContext.

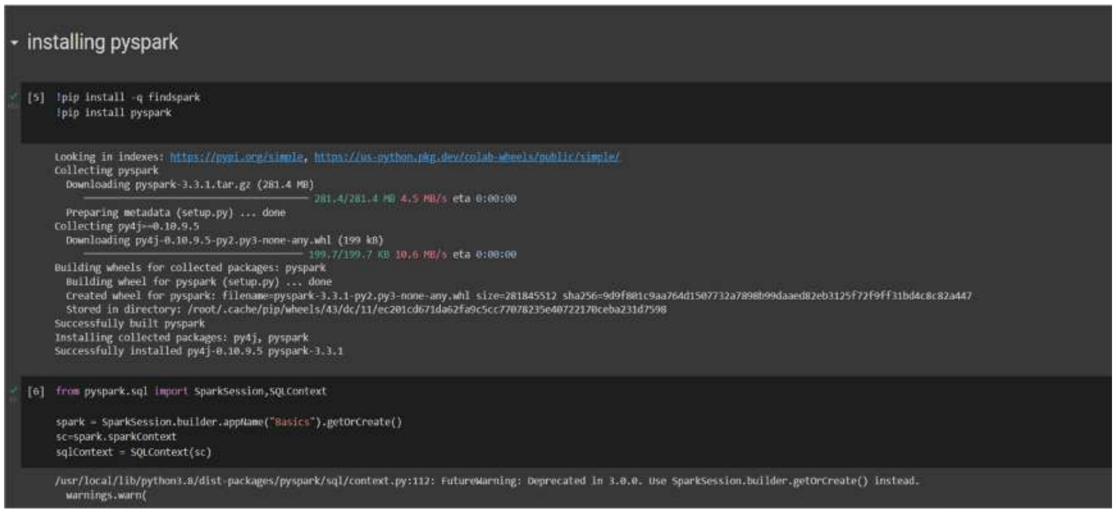
Be default PySpark shell provides “spark” object; which is an instance of SparkSession class. We can directly use this object where required in spark-shell. Start your “pyspark” shell from **$SPARK_HOME\bin** folder and enter the pyspark command. Once you are in the PySpark shell enter the below command to get the PySpark version.



```
# Usage of spark object in PySpark shell
>>>spark.version
3.1.2
```



```
Python 3.9.7 (default, Sep 16 2021, 08:50:36)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.apache.spark.unsafe.Platform (file:/Users/admin/opt/anaconda3/lib/python3.9/site
-packages/pyspark/jars/spark-unsafe_2.12-3.1.2.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
22/05/24 00:16:38 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java clas
ses where applicable
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).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.1.2
      /_/

Using Python version 3.9.7 (default, Sep 16 2021 08:50:36)
Spark context Web UI available at http://naveens-mbp.attlocal.net:4040
Spark context available as 'sc' (master = local[*], app id = local-1653376599050).
SparkSession available as 'spark'.
>>> spark.version
'3.1.2'
>>> 
```



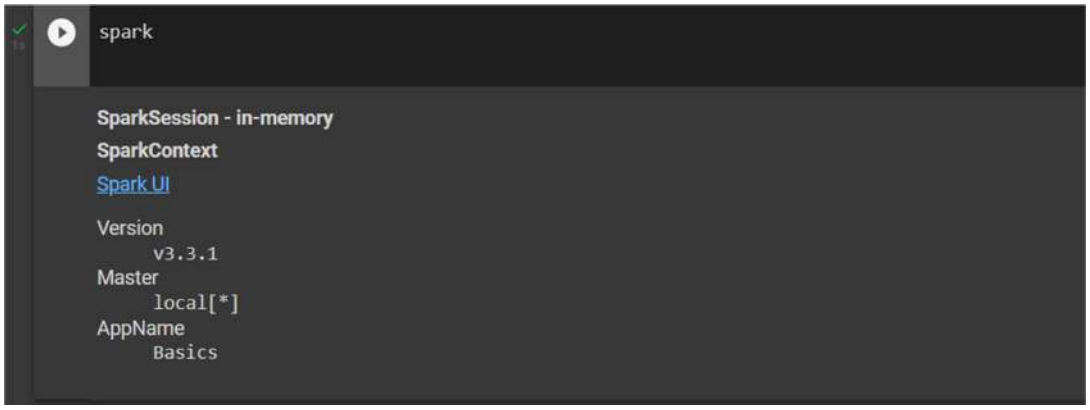
installing pyspark

```
[5]  !pip install -q findspark
     !pip install pyspark

     Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
     Collecting pyspark
       Downloading pyspark-3.3.1.tar.gz (281.4 MB)
          ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 281.4/281.4 MB 4.5 MB/s eta 0:00:00
       Preparing metadata (setup.py) ... done
     Collecting py4j==0.10.9.5
       Downloading py4j-0.10.9.5-py2.py3-none-any.whl (199 kB)
          ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 199.7/199.7 KB 10.6 MB/s eta 0:00:00
     Building wheels for collected packages: pyspark
       Building wheel for pyspark (setup.py) ... done
       Created wheel for pyspark: filename=pyspark-3.3.1-py2.py3-none-any.whl size=281845512 sha256=9d9f801c9aa764d1507732a7898b99daaed82eb3125f72f9ff31bd4c8c82a447
       Stored in directory: /root/.cache/pip/wheels/43/dc/11/ec201cd671da62fa9c5cc77078235e40722170ceba231d7598
     Successfully built pyspark
     Installing collected packages: py4j, pyspark
     Successfully installed py4j-0.10.9.5 pyspark-3.3.1

[6]  from pyspark.sql import SparkSession,SQLContext

     spark = SparkSession.builder.appName("Basics").getOrCreate()
     sc=spark.sparkContext
     sqlContext = SQLContext(sc)

     /usr/local/lib/python3.8/dist-packages/pyspark/sql/context.py:112: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead.
       warnings.warn(
```

For installing pyspark **!pip install –q findspark** and **!pip install pyspark** commands are given which installs and loads the necessary packages py4j and pyspark. For creating a SparkSession from builder we imported SparkSession from pyspark.sql. getOrCreate() returns already existing SparkSession. appName() is used to set your application name.

Section 2.2. Pyspark- SparkContext

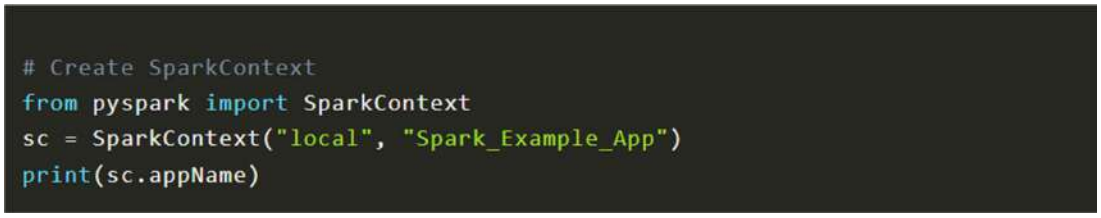
Pyspark.SparkContext is an entry point to the PySpark functionality that is used to communicate with the cluster and to create an RDD, accumulator, and broadcast variables. "spark" command is given to check the version of the spark used and the appName.



```
spark
```

```
SparkSession - in-memory
SparkContext
Spark UI

Version
    v3.3.1
Master
    local[*]
AppName
    Basics
```

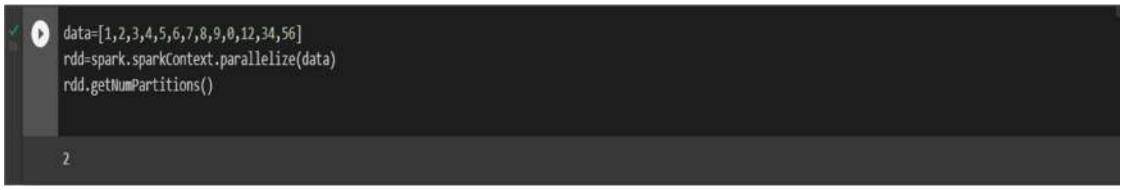
For creating a SparkContext prior to Pyspark 2.0 by using a constructor and passes parameters like master() and appName(), SparkContext.getOrCreate() is also used for returning an existing active SparkContext. The Spark driver program creates and uses SparkContext to connect to the cluster manager to submit PySpark jobs, and know what resource manager (YARN, Mesos, or Standalone) to communicate to.



```
# Create SparkContext
from pyspark import SparkContext
sc = SparkContext("local", "Spark_Example_App")
print(sc.appName)
```

Section 2.3. Pyspark- RDD

RDD (Resilient Distributed Dataset) is a fundamental building block of pyspark which is fault-tolerant and has immutable distributed collections of objects which means that once RDD is created you cannot change it. RDD is divided into number of logical partitions which can be computed on different nodes of cluster. RDDs are collection of objects similar to list in python it provides data abstraction of partitioning and distribution of data which are designed to run computations on several nodes parallel. Basic operations available on RDD's are map(), filter(), persist() and may more.

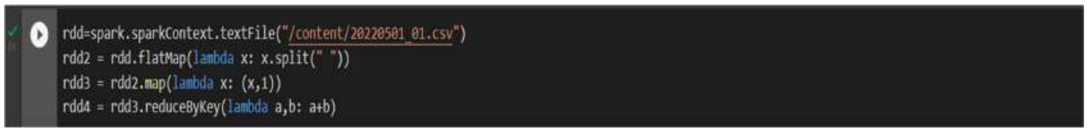


```
data=[1,2,3,4,5,6,7,8,9,0,12,34,56]
rdd=spark.sparkContext.parallelize(data)
rdd.getNumPartitions()

2
```

When we use **parallelize()** or **textFiles()** or **wholeTextFiles()** methods of SparkContext, it automatically splits the data into partitions based on the resource availability and would create the partitions as the number of cores available in your system. getNumPartitions() is used to display the count of partitions created depending upon the data.

OPERATIONS OF RDD:

- RDD Transformations: RDD transformations return another RDD and they don’t execute until an action is called on RDD. Some transformations on RDDs are flatMap(), reduceByKey(), filter(), map(), sortByKey() and returns a new RDD instead of updating the current.



```
rdd=spark.sparkContext.textFile("/content/20220501_01.csv")
rdd2 = rdd.flatMap(lambda x: x.split(" "))
rdd3 = rdd2.map(lambda x: (x,1))
rdd4 = rdd3.reduceByKey(lambda a,b: a+b)
```

i. flatMap() flattens the RDD created and returns the new RDD. First it splits the data and then flattens it.
ii. map() transformations is always used for the complex operations such as a adding the column, updating the column, etc. Output of map() transformations will always have the same number of records as the input.
iii. reduceByKey() merges the value for each key according to the function applied.

- RDD ACTIONS: RDD actions returns the value from an RDD to a driver program. In other words, any RDD that returns a non-RDD is considered as the action. Some RDD actions include count(), first(), max(), reduce(), take(), collect().

i. count() returns the number of records.
ii. first() returns the first record.
iii. max() returns the maximum record.

iv. reduce() reduces the records into single. We can use to evaluate count or sum.
v. collect() returns all the data from an RDD as an array.

BENEFITS OF PYSPARK RDD:

- *Immutability:* PySpark RDD's are immutable in nature meaning, once RDDs are created you cannot modify. When we apply transformations on RDD, PySpark creates a new RDD and maintains the RDD Lineage.

- *Fault tolerance:* PySpark operates on fault-tolerant data stores on HDFS, S3 etc. hence any RDD operation fails, it automatically reloads the data from other partitions. Also, When PySpark applications running on a cluster, PySpark task failures are automatically recovered for a certain number of times (as per the configuration) and finish the application seamlessly.

- *Lazy evaluation:* PySpark does not evaluate the RDD transformations as they appear/encountered by Driver instead it keeps the all transformations as it encounters (DAG) and evaluates the all transformation when it sees the first RDD action.

- *Partitioning:* When you create RDD from a data, it by default partitions the elements in a RDD. By default it partitions to the number of cores available.

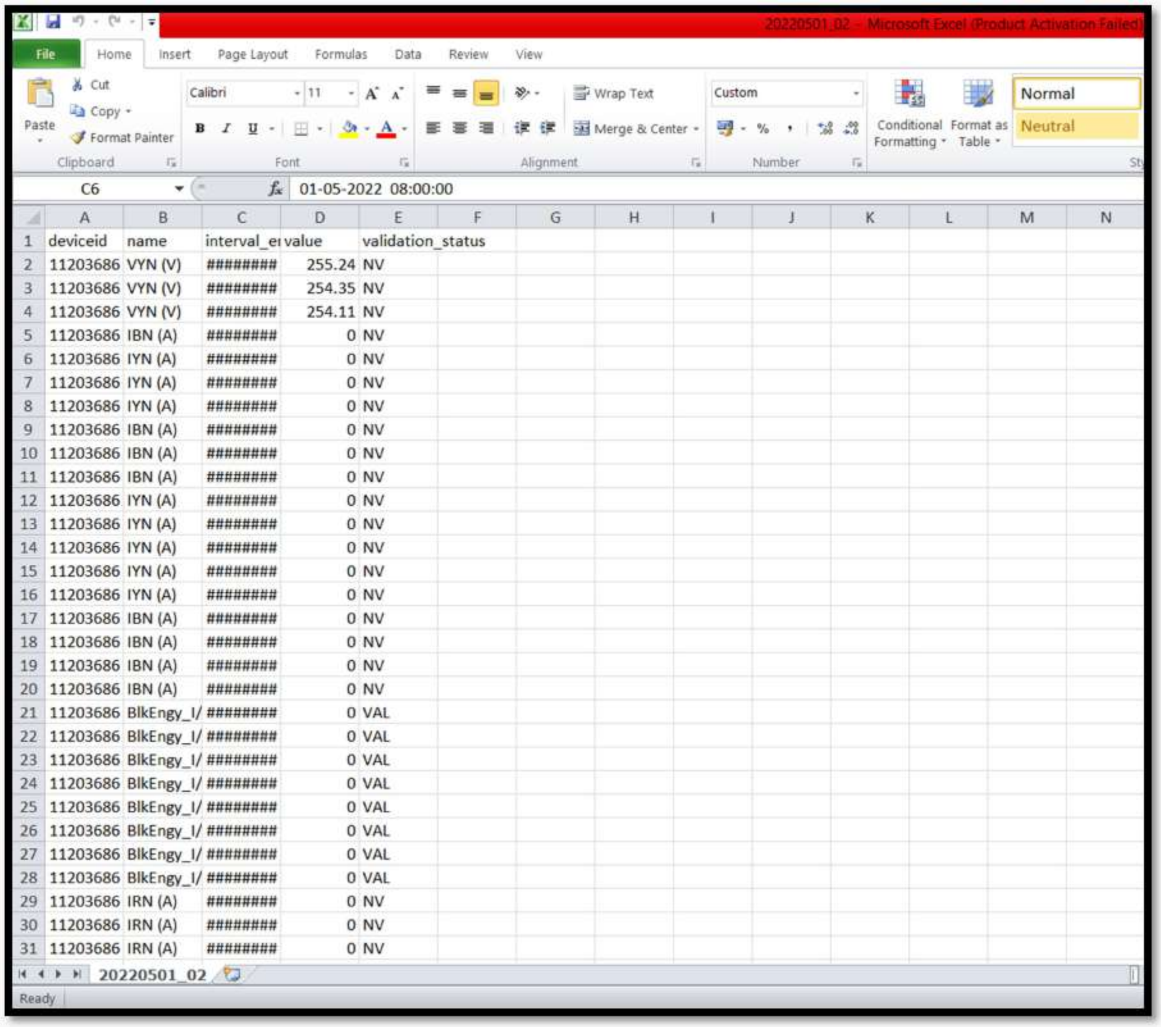
Section 3.0. Reading and Writing Datasets

PySpark provides csv("path") on DataFrameReader to read a CSV file into pyspark Data Frame and dataframeobj.write.csv("path") to save or write to the CSV file. In this tutorial, you will learn how to read a single file, multiple files, all files from a local directory into Data Frame, applying some transformations, and finally writing Data Frame back to CSV file PySpark supports reading a CSV file with a pipe, comma, tab, space, or any other delimiter/separator files.

Section 3.1. CSV file

Using csv("path") or format("csv").load("path") of DataFrameReader, you can read a CSV file into a PySpark Data Frame, These methods take a file path to read from as an argument. When you use format("csv") method, you can also specify the Data sources by their fully qualified name, but for built-in sources, you can simply use their short names (csv, json, parquet, text etc.) The csv text-file for the proactive voltage has been loaded. The dataset of smart meter voltage was loaded, consisting of the device-id, interval end time and the validation status along with the values of each device at different intervals were given. The two datasets can be seen in the excel, for the visualization and analysis of the large data of smart-meter can be done with the help of the operations and functions of the Pyspark.

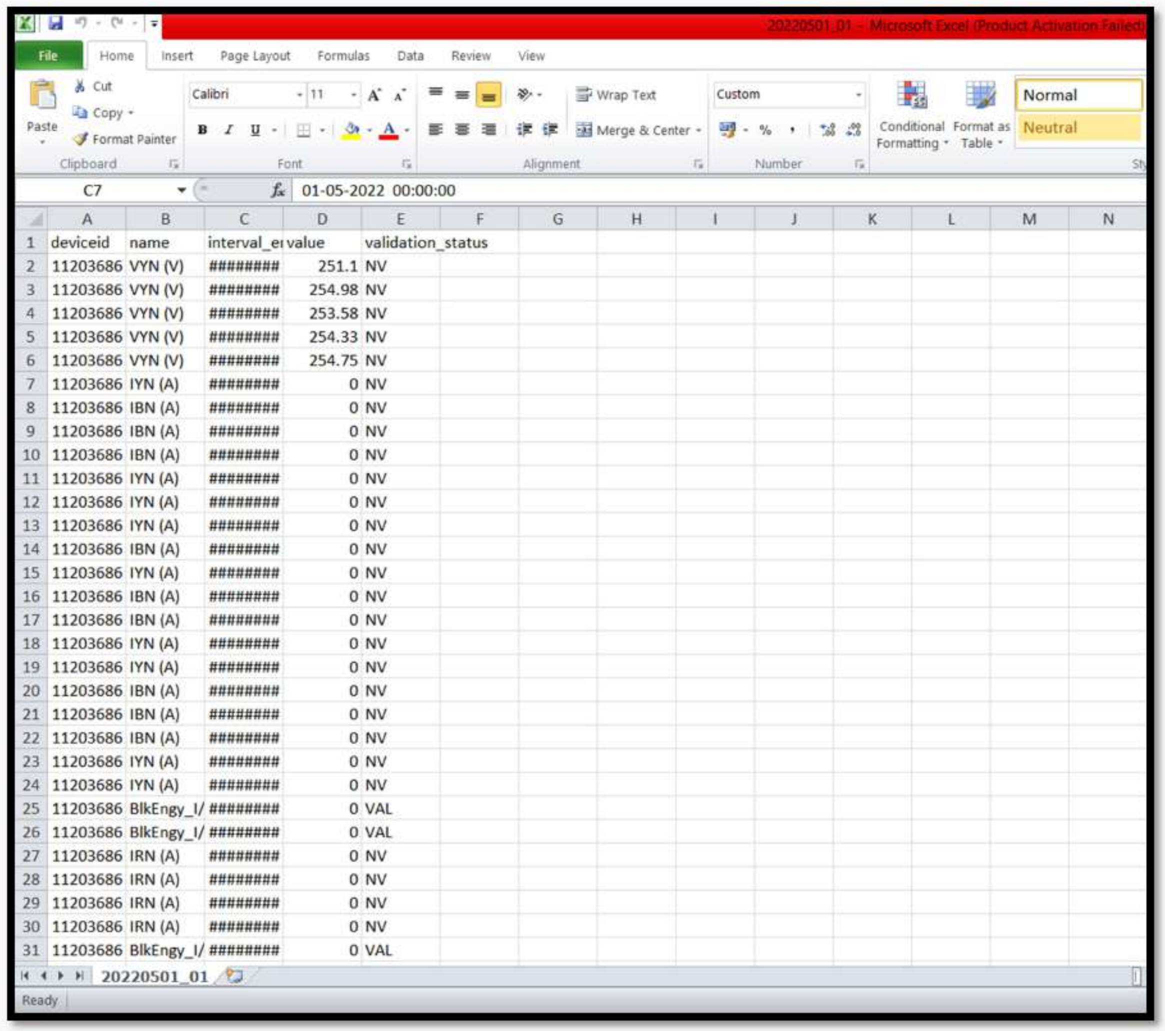
Firstly the csv format data is loaded after installing the pyspark by giving the installation commands. The dataset can be converted into pandas format or in rdd form and can also be changed to the parquet format by performing some of the writing and reading commands.



C6 fx 01-05-2022 08:00:00

	deviceid	name	interval_e	value	validation_status
2	11203686	VYN (V)	########	255.24	NV
3	11203686	VYN (V)	########	254.35	NV
4	11203686	VYN (V)	########	254.11	NV
5	11203686	IBN (A)	########	0	NV
6	11203686	IYN (A)	########	0	NV
7	11203686	IYN (A)	########	0	NV
8	11203686	IYN (A)	########	0	NV
9	11203686	IBN (A)	########	0	NV
10	11203686	IBN (A)	########	0	NV
11	11203686	IBN (A)	########	0	NV
12	11203686	IYN (A)	########	0	NV
13	11203686	IYN (A)	########	0	NV
14	11203686	IYN (A)	########	0	NV
15	11203686	IYN (A)	########	0	NV
16	11203686	IYN (A)	########	0	NV
17	11203686	IBN (A)	########	0	NV
18	11203686	IBN (A)	########	0	NV
19	11203686	IBN (A)	########	0	NV
20	11203686	IBN (A)	########	0	NV
21	11203686	BlkEngy_I/	########	0	VAL
22	11203686	BlkEngy_I/	########	0	VAL
23	11203686	BlkEngy_I/	########	0	VAL
24	11203686	BlkEngy_I/	########	0	VAL
25	11203686	BlkEngy_I/	########	0	VAL
26	11203686	BlkEngy_I/	########	0	VAL
27	11203686	BlkEngy_I/	########	0	VAL
28	11203686	BlkEngy_I/	########	0	VAL
29	11203686	IRN (A)	########	0	NV
30	11203686	IRN (A)	########	0	NV
31	11203686	IRN (A)	########	0	NV

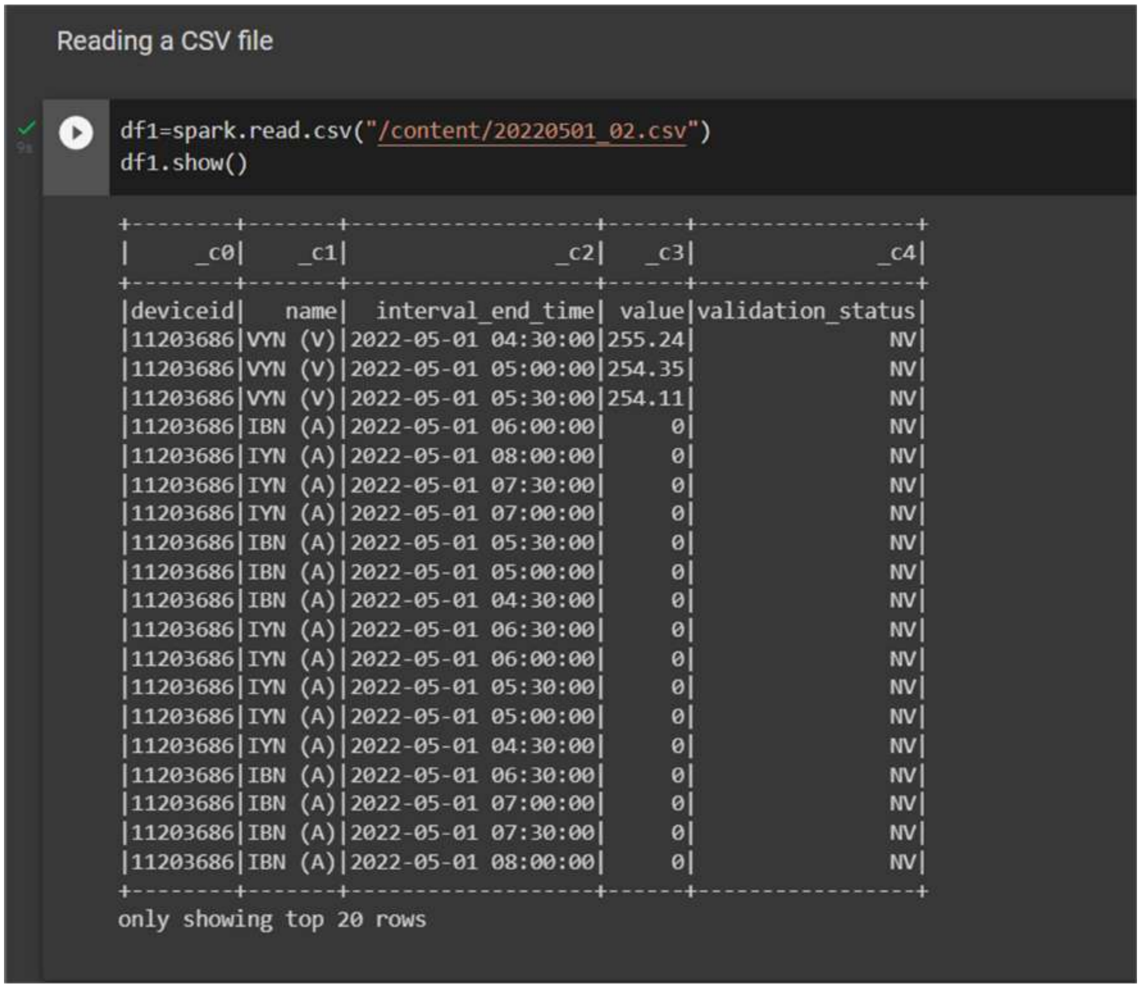
Figure 1. Smart-meter dataset 1



01-05-2022 00:00:00

	A	B	C	D	E
1	deviceid	name	interval_er	value	validation_status
2	11203686	VYN (V)	########	251.1	NV
3	11203686	VYN (V)	########	254.98	NV
4	11203686	VYN (V)	########	253.58	NV
5	11203686	VYN (V)	########	254.33	NV
6	11203686	VYN (V)	########	254.75	NV
7	11203686	IYN (A)	########	0	NV
8	11203686	IBN (A)	########	0	NV
9	11203686	IBN (A)	########	0	NV
10	11203686	IBN (A)	########	0	NV
11	11203686	IYN (A)	########	0	NV
12	11203686	IYN (A)	########	0	NV
13	11203686	IYN (A)	########	0	NV
14	11203686	IBN (A)	########	0	NV
15	11203686	IYN (A)	########	0	NV
16	11203686	IBN (A)	########	0	NV
17	11203686	IBN (A)	########	0	NV
18	11203686	IYN (A)	########	0	NV
19	11203686	IYN (A)	########	0	NV
20	11203686	IBN (A)	########	0	NV
21	11203686	IBN (A)	########	0	NV
22	11203686	IBN (A)	########	0	NV
23	11203686	IYN (A)	########	0	NV
24	11203686	IYN (A)	########	0	NV
25	11203686	BlkEngy_I/	########	0	VAL
26	11203686	BlkEngy_I/	########	0	VAL
27	11203686	IRN (A)	########	0	NV
28	11203686	IRN (A)	########	0	NV
29	11203686	IRN (A)	########	0	NV
30	11203686	IRN (A)	########	0	NV
31	11203686	BlkEngy_I/	########	0	VAL

Figure 2. Smart-meter dataset 2

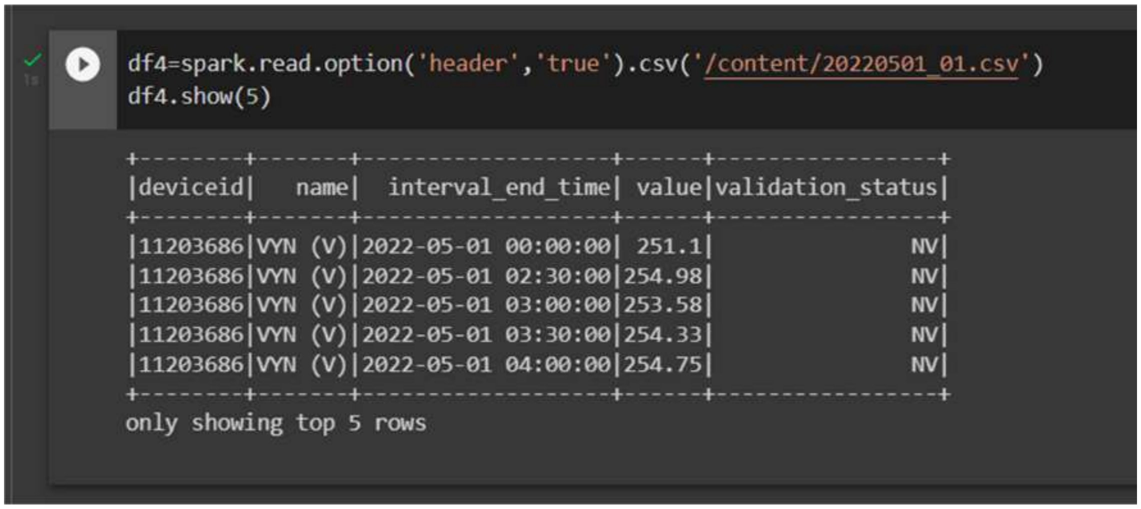

Reading a CSV file

```
df1=spark.read.csv("/content/20220501_02.csv")
df1.show()
```

```
+--------+-------+-------------------+------+-----------------+
|     _c0|    _c1|                _c2|   _c3|              _c4|
+--------+-------+-------------------+------+-----------------+
|deviceid|   name|  interval_end_time| value|validation_status|
|11203686|VYN (V)|2022-05-01 04:30:00|255.24|               NV|
|11203686|VYN (V)|2022-05-01 05:00:00|254.35|               NV|
|11203686|VYN (V)|2022-05-01 05:30:00|254.11|               NV|
|11203686|IBN (A)|2022-05-01 06:00:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 08:00:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 07:30:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 07:00:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 05:30:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 05:00:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 04:30:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 06:30:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 06:00:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 05:30:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 05:00:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 04:30:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 06:30:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 07:00:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 07:30:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 08:00:00|     0|               NV|
+--------+-------+-------------------+------+-----------------+
only showing top 20 rows
```

Figure 3. Pyspark command for reading a csv file

But in this we can see that after reading the csv file the column names are by default given as "_c0", "_c1" and so on, therefore to change this the command is modified and given as shown in figure 4.



```
df4=spark.read.option('header','true').csv('/content/20220501_01.csv')
df4.show(5)
```

```
+--------+-------+-------------------+------+-----------------+
|deviceid|   name|  interval_end_time| value|validation_status|
+--------+-------+-------------------+------+-----------------+
|11203686|VYN (V)|2022-05-01 00:00:00| 251.1|               NV|
|11203686|VYN (V)|2022-05-01 02:30:00|254.98|               NV|
|11203686|VYN (V)|2022-05-01 03:00:00|253.58|               NV|
|11203686|VYN (V)|2022-05-01 03:30:00|254.33|               NV|
|11203686|VYN (V)|2022-05-01 04:00:00|254.75|               NV|
+--------+-------+-------------------+------+-----------------+
only showing top 5 rows
```

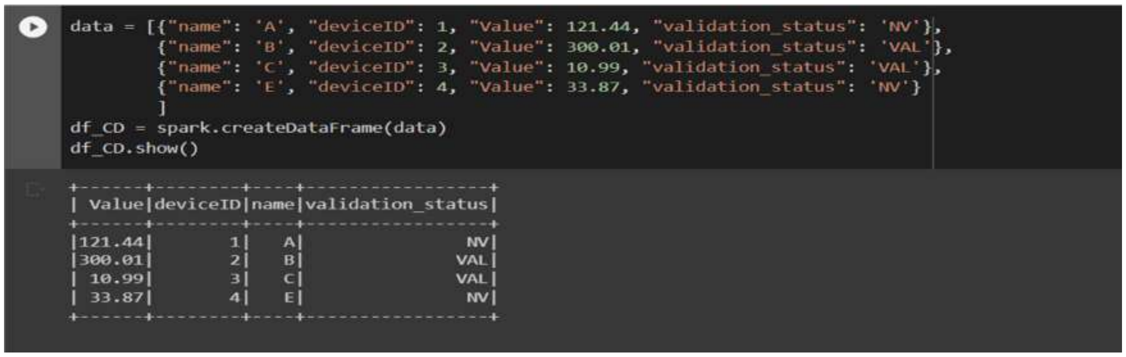
Figure 4. Modified command for reading

show() command is used to display the dataframe. Using read.csv() file we can also read multiple csv files by adding the paths of all the files separated by the commas.

```
df = spark.read.csv("path1,path2,path3")
```

Section 3.2. Creating a Dataframe

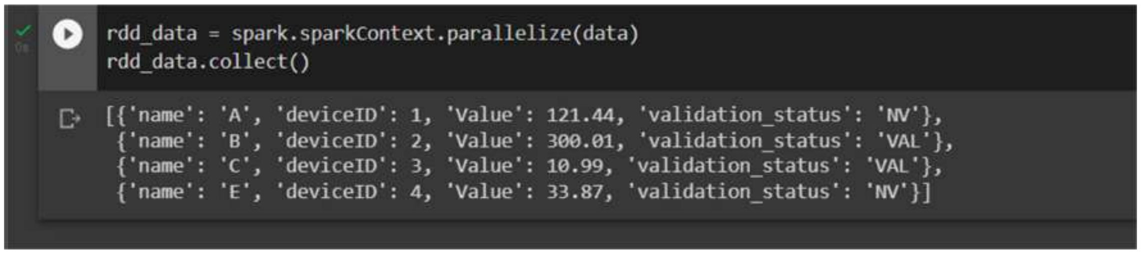
You can create DataFrame using toDF() and createDataFrame() methods, both these function takes different signatures in order to create DataFrame from existing RDD, list, and DataFrame.
You can also create PySpark DataFrame from data sources like TXT, CSV, JSON, ORV, Avro, Parquet, XML formats by reading from HDFS, S3, DBFS, Azure Blob file systems etc.



```
data = [{"name": 'A', "deviceID": 1, "Value": 121.44, "validation_status": 'NV'},
        {"name": 'B', "deviceID": 2, "Value": 300.01, "validation_status": 'VAL'},
        {"name": 'C', "deviceID": 3, "Value": 10.99, "validation_status": 'VAL'},
        {"name": 'E', "deviceID": 4, "Value": 33.87, "validation_status": 'NV'}
        ]
df_CD = spark.createDataFrame(data)
df_CD.show()
```

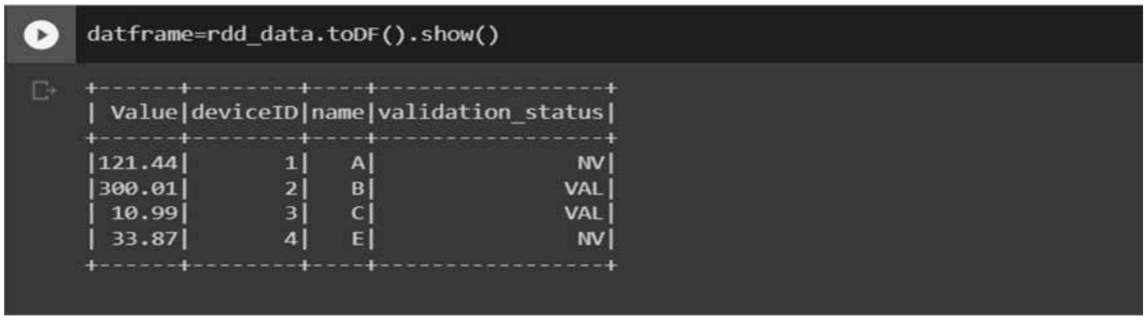
```
+------+--------+----+-----------------+
| Value|deviceID|name|validation_status|
+------+--------+----+-----------------+
|121.44|       1|   A|               NV|
|300.01|       2|   B|              VAL|
| 10.99|       3|   C|              VAL|
| 33.87|       4|   E|               NV|
+------+--------+----+-----------------+
```

With the help of parallelize command we can convert we can create Pyspark RDD.



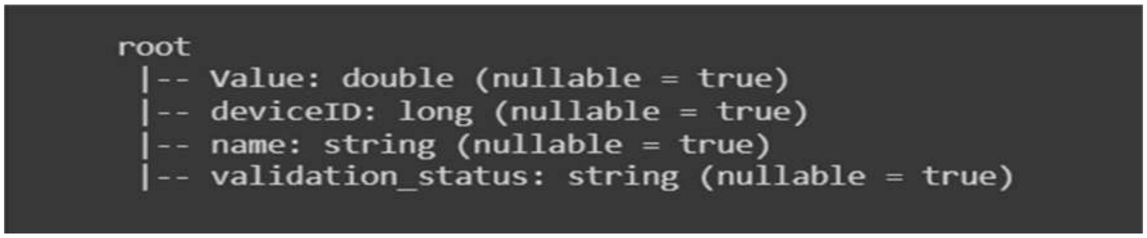
```
rdd_data = spark.sparkContext.parallelize(data)
rdd_data.collect()

[{'name': 'A', 'deviceID': 1, 'Value': 121.44, 'validation_status': 'NV'},
 {'name': 'B', 'deviceID': 2, 'Value': 300.01, 'validation_status': 'VAL'},
 {'name': 'C', 'deviceID': 3, 'Value': 10.99, 'validation_status': 'VAL'},
 {'name': 'E', 'deviceID': 4, 'Value': 33.87, 'validation_status': 'NV'}]
```

By using toDF() we can convert RDD to Dataframe again.



```
datframe=rdd_data.toDF().show()

+------+--------+----+-----------------+
| Value|deviceID|name|validation_status|
+------+--------+----+-----------------+
|121.44|       1|   A|               NV|
|300.01|       2|   B|              VAL|
| 10.99|       3|   C|              VAL|
| 33.87|       4|   E|               NV|
+------+--------+----+-----------------+
```

printSchema() commands prints the type of the column variables:



```
root
 |-- Value: double (nullable = true)
 |-- deviceID: long (nullable = true)
 |-- name: string (nullable = true)
 |-- validation_status: string (nullable = true)
```

Section 3.3. What is Parquet

Parquet is an open source file format built to handle flat columnar storage data formats. Parquet operates well with complex data in large volumes. It is known for its both performance data compression and its ability to handle a wide variety of encoding types.

<u>HOW PARQUET IS DIFFERENT FROM CSV FILE FORMAL:</u>

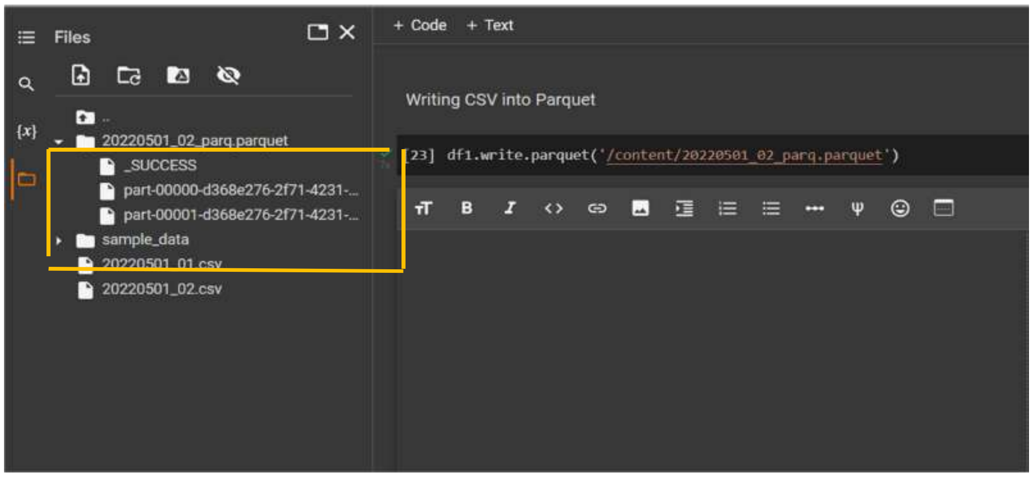
While CSV is simple and the most widely used data format (Excel, Google Sheets), there are several distinct advantages for Parquet, including:

- Parquet is column oriented and CSV is row oriented. Row-oriented formats are optimized for OLTP workloads while column-oriented formats are better suited for analytical workloads.
- Column-oriented databases such as AWS Redshift Spectrum bill by the amount data scanned per query
- Therefore, converting CSV to Parquet with partitioning and compression lowers overall costs and improves performance

Parquet has helped its users reduce storage requirements by at least one-third on large datasets, in addition, it greatly improves scan and deserialization time, hence the overall costs.

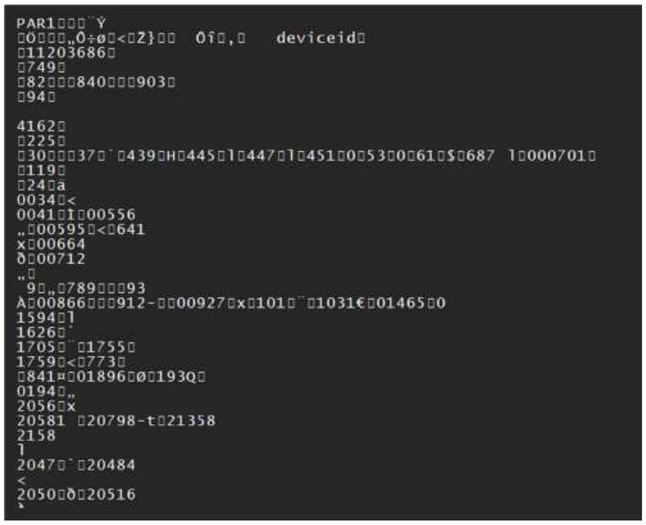
Secton 3.4. Writing CSV to Parquet

We can convert any dataframe or csv file format into parquet format.



```
[23] df1.write.parquet('/content/20220501_02_parq.parquet')
```

Figure 1. CSV dataset is written in parquet format

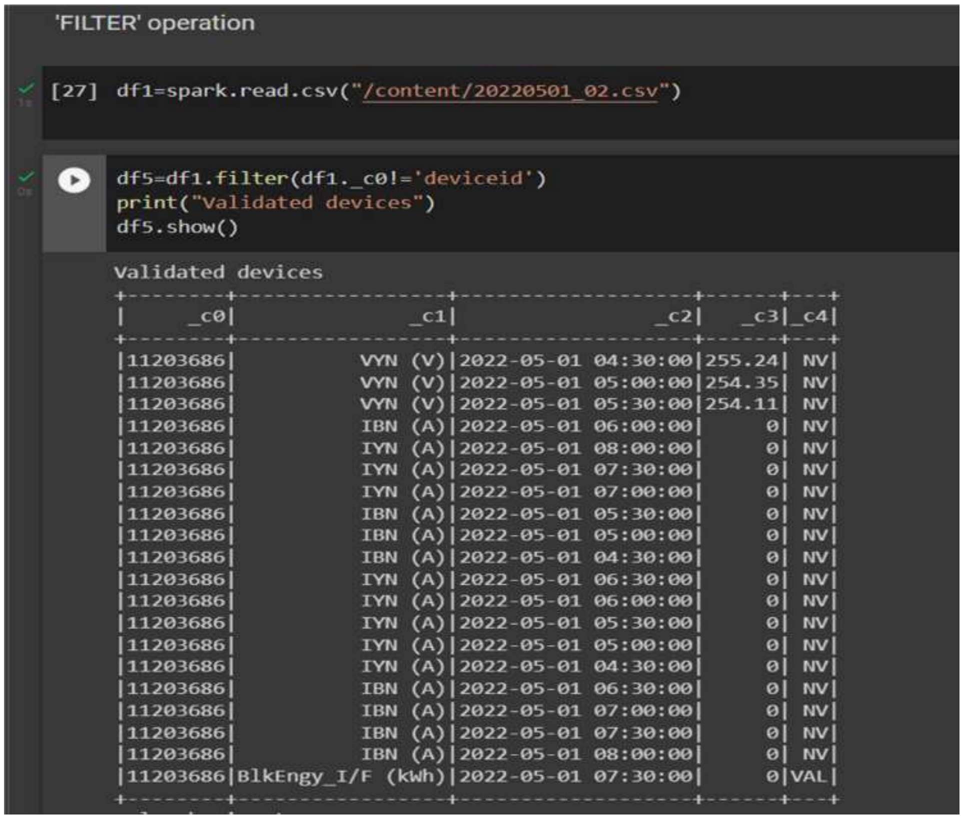


```
PAR1▯▯▯¨Ÿ
▯Ö▯▯▯„Ô÷ø▯<▯Z}▯▯  Ôî▯,▯   deviceid▯
▯11203686▯
▯749▯
▯82▯▯▯840▯▯▯903▯
▯94▯

4162▯
▯225▯
▯30▯▯▯37▯`▯439▯H▯445▯1▯447▯1▯451▯0▯53▯0▯61▯$▯687 1▯000701▯
▯119▯
▯24▯ä
0034▯<
0041▯Ì▯00556
„▯00595▯<▯641
x▯00664
ð▯00712
„▯
 9▯„▯789▯▯▯93
À▯00866▯▯▯912-▯▯00927▯x▯101▯¨▯1031€▯01465▯0
1594▯1
1626▯`
1705▯¨▯1755▯
1759▯<▯773▯
▯841¤▯01896▯Ø▯193Q▯
0194▯„
2056▯x
20581 ▯20798-t▯21358
2158
1
2047▯`▯20484
<
2050▯ð▯20516
`
```

Figure 2. Parquet file output

Section 4.0. Commands for performing analysis on dataset

- **'FILTER' Command:** PySpark filter() function is used to filter the rows from RDD/DataFrame based on the given condition or SQL expression, you can also use where() clause instead of the filter() if you are coming from an SQL background, both these functions operate exactly the same. Below are two examples of the filter() operation used on both the datasets of smart meter. To filter those devices that are validated.


'FILTER' operation

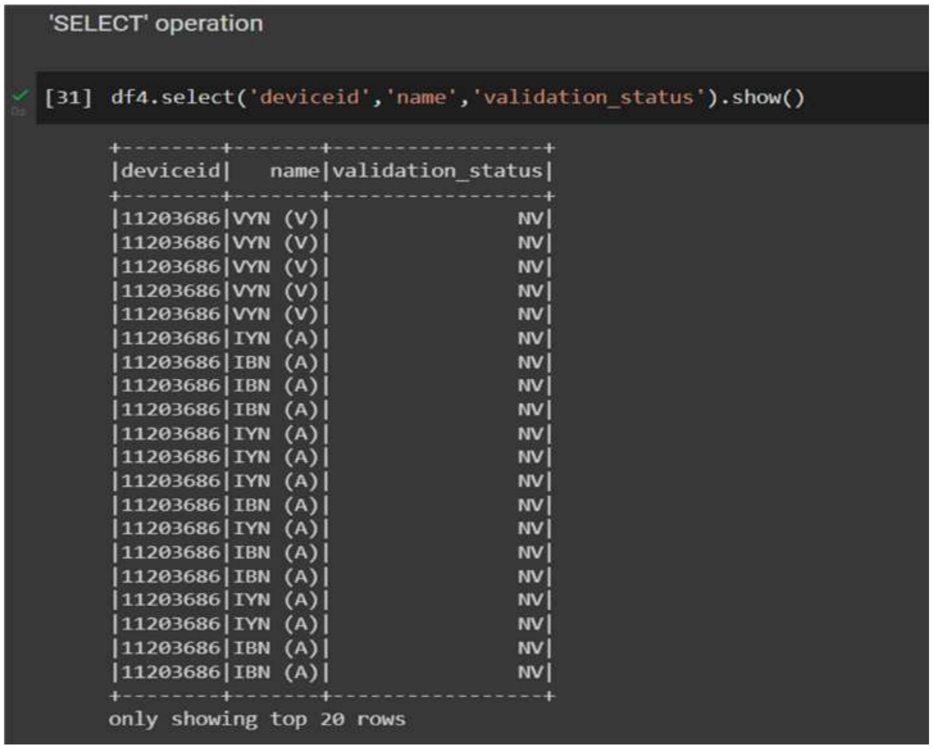
```
[27] df1=spark.read.csv("/content/20220501_02.csv")
```

```
df5=df1.filter(df1._c0!='deviceid')
print("Validated devices")
df5.show()
```

```
Validated devices
+--------+-----------------+-------------------+------+---+
|     _c0|              _c1|                _c2|   _c3|_c4|
+--------+-----------------+-------------------+------+---+
|11203686|          VYN (V)|2022-05-01 04:30:00|255.24| NV|
|11203686|          VYN (V)|2022-05-01 05:00:00|254.35| NV|
|11203686|          VYN (V)|2022-05-01 05:30:00|254.11| NV|
|11203686|          IBN (A)|2022-05-01 06:00:00|     0| NV|
|11203686|          IYN (A)|2022-05-01 08:00:00|     0| NV|
|11203686|          IYN (A)|2022-05-01 07:30:00|     0| NV|
|11203686|          IYN (A)|2022-05-01 07:00:00|     0| NV|
|11203686|          IBN (A)|2022-05-01 05:30:00|     0| NV|
|11203686|          IBN (A)|2022-05-01 05:00:00|     0| NV|
|11203686|          IBN (A)|2022-05-01 04:30:00|     0| NV|
|11203686|          IYN (A)|2022-05-01 06:30:00|     0| NV|
|11203686|          IYN (A)|2022-05-01 06:00:00|     0| NV|
|11203686|          IYN (A)|2022-05-01 05:30:00|     0| NV|
|11203686|          IYN (A)|2022-05-01 05:00:00|     0| NV|
|11203686|          IYN (A)|2022-05-01 04:30:00|     0| NV|
|11203686|          IBN (A)|2022-05-01 06:30:00|     0| NV|
|11203686|          IBN (A)|2022-05-01 07:00:00|     0| NV|
|11203686|          IBN (A)|2022-05-01 07:30:00|     0| NV|
|11203686|          IBN (A)|2022-05-01 08:00:00|     0| NV|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 07:30:00|     0|VAL|
+--------+-----------------+-------------------+------+---+
```

Figure 1. Filtering the validated devices

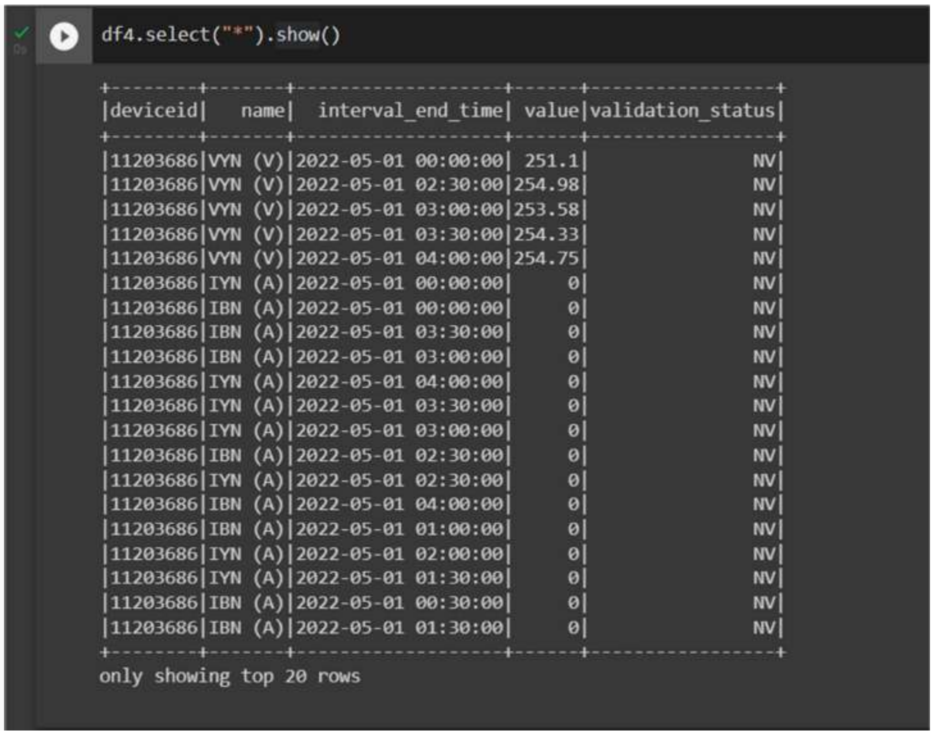
- **'SELECT' Command :** In PySpark, select() function is used to select single, multiple, column by index, all columns from the list and the nested columns from a DataFrame, PySpark select() is a transformation function hence it returns a new DataFrame with the selected columns.
  If we have to analyse the validation status of the devices and we don't need the 'value' column then we can use the select() command to see the validation status of each device corresponding to their names and ID. Top 20 rows are displayed by default.



'SELECT' operation

```
[31] df4.select('deviceid','name','validation_status').show()

+--------+-------+-----------------+
|deviceid|   name|validation_status|
+--------+-------+-----------------+
|11203686|VYN (V)|               NV|
|11203686|VYN (V)|               NV|
|11203686|VYN (V)|               NV|
|11203686|VYN (V)|               NV|
|11203686|VYN (V)|               NV|
|11203686|IYN (A)|               NV|
|11203686|IBN (A)|               NV|
|11203686|IBN (A)|               NV|
|11203686|IBN (A)|               NV|
|11203686|IYN (A)|               NV|
|11203686|IYN (A)|               NV|
|11203686|IYN (A)|               NV|
|11203686|IBN (A)|               NV|
|11203686|IYN (A)|               NV|
|11203686|IBN (A)|               NV|
|11203686|IBN (A)|               NV|
|11203686|IYN (A)|               NV|
|11203686|IYN (A)|               NV|
|11203686|IBN (A)|               NV|
|11203686|IBN (A)|               NV|
+--------+-------+-----------------+
only showing top 20 rows
```

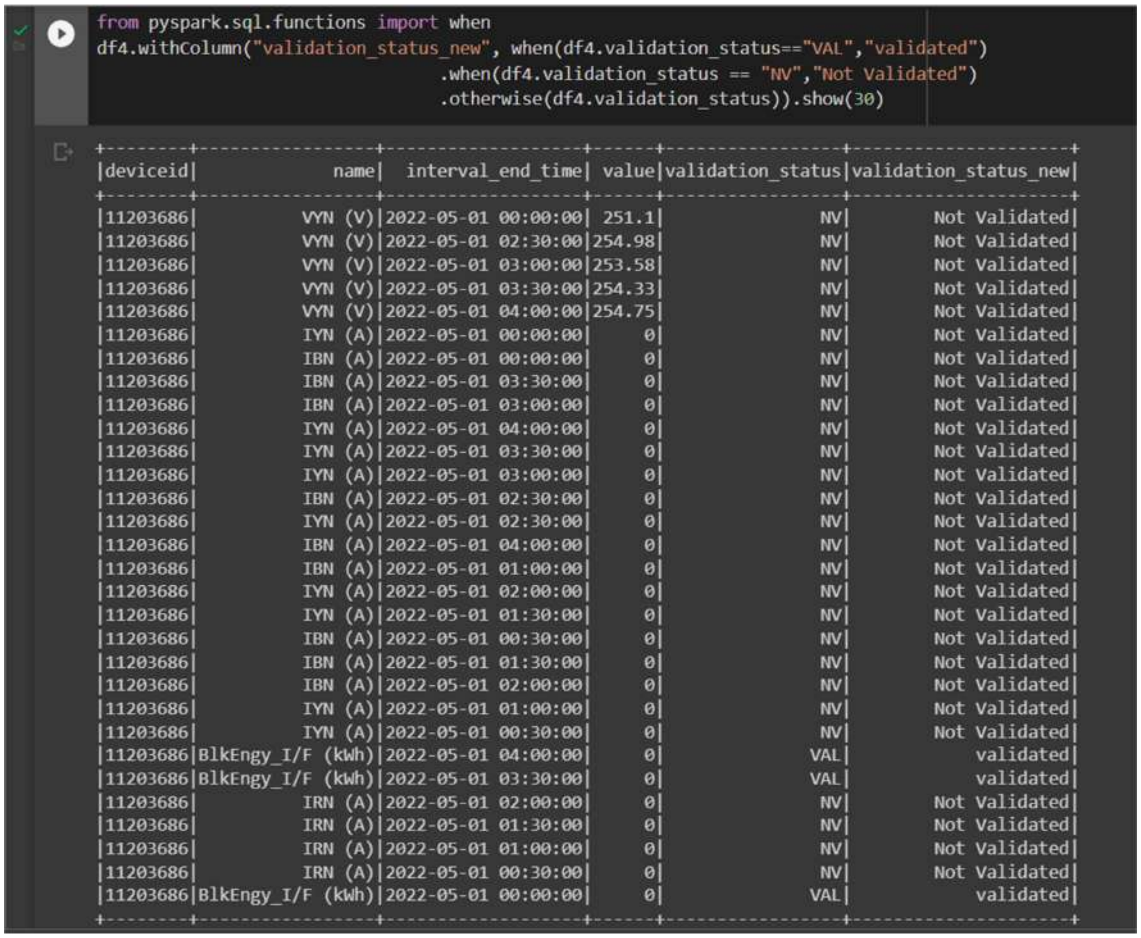
Figure 2.0. Selecting multiple columns from the dataset



```
df4.select("*").show()

+--------+-------+-------------------+------+-----------------+
|deviceid|   name|  interval_end_time| value|validation_status|
+--------+-------+-------------------+------+-----------------+
|11203686|VYN (V)|2022-05-01 00:00:00| 251.1|               NV|
|11203686|VYN (V)|2022-05-01 02:30:00|254.98|               NV|
|11203686|VYN (V)|2022-05-01 03:00:00|253.58|               NV|
|11203686|VYN (V)|2022-05-01 03:30:00|254.33|               NV|
|11203686|VYN (V)|2022-05-01 04:00:00|254.75|               NV|
|11203686|IYN (A)|2022-05-01 00:00:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 00:00:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 03:30:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 03:00:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 04:00:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 03:30:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 03:00:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 02:30:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 02:30:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 04:00:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 01:00:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 02:00:00|     0|               NV|
|11203686|IYN (A)|2022-05-01 01:30:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 00:30:00|     0|               NV|
|11203686|IBN (A)|2022-05-01 01:30:00|     0|               NV|
+--------+-------+-------------------+------+-----------------+
only showing top 20 rows
```

Figure 2.1. Select("") can also be used to display all columns*

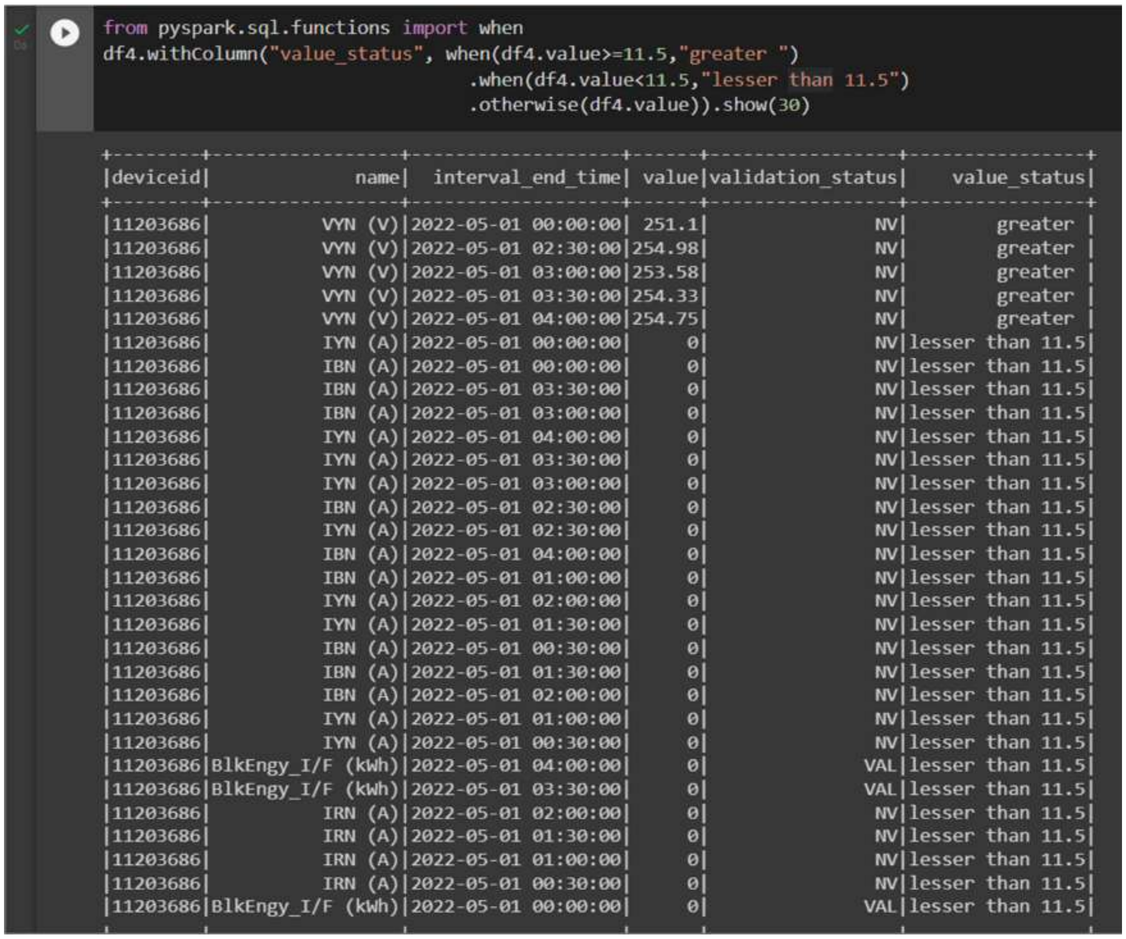
- **When() and otherwise on Pyspark Data Frame:** PySpark when() is SQL function, in order to use this first you should import and this returns a Column type, otherwise() is a function of column, when otherwise() not used and none of the conditions met it assigns None (Null) value. when() function takes 2 parameters, first parameter takes a condition and second takes a literal value or Column, if condition evaluates to true then it returns a value from second parameter.



```
from pyspark.sql.functions import when
df4.withColumn("validation_status_new", when(df4.validation_status=="VAL","validated")
                                   .when(df4.validation_status == "NV","Not Validated")
                                   .otherwise(df4.validation_status)).show(30)
```

```
+--------+----------------+-------------------+------+-----------------+---------------------+
|deviceid|            name|  interval_end_time| value|validation_status|validation_status_new|
+--------+----------------+-------------------+------+-----------------+---------------------+
|11203686|         VYN (V)|2022-05-01 00:00:00| 251.1|               NV|        Not Validated|
|11203686|         VYN (V)|2022-05-01 02:30:00|254.98|               NV|        Not Validated|
|11203686|         VYN (V)|2022-05-01 03:00:00|253.58|               NV|        Not Validated|
|11203686|         VYN (V)|2022-05-01 03:30:00|254.33|               NV|        Not Validated|
|11203686|         VYN (V)|2022-05-01 04:00:00|254.75|               NV|        Not Validated|
|11203686|         IYN (A)|2022-05-01 00:00:00|     0|               NV|        Not Validated|
|11203686|         IBN (A)|2022-05-01 00:00:00|     0|               NV|        Not Validated|
|11203686|         IBN (A)|2022-05-01 03:30:00|     0|               NV|        Not Validated|
|11203686|         IBN (A)|2022-05-01 03:00:00|     0|               NV|        Not Validated|
|11203686|         IYN (A)|2022-05-01 04:00:00|     0|               NV|        Not Validated|
|11203686|         IYN (A)|2022-05-01 03:30:00|     0|               NV|        Not Validated|
|11203686|         IYN (A)|2022-05-01 03:00:00|     0|               NV|        Not Validated|
|11203686|         IBN (A)|2022-05-01 02:30:00|     0|               NV|        Not Validated|
|11203686|         IYN (A)|2022-05-01 02:30:00|     0|               NV|        Not Validated|
|11203686|         IBN (A)|2022-05-01 04:00:00|     0|               NV|        Not Validated|
|11203686|         IBN (A)|2022-05-01 01:00:00|     0|               NV|        Not Validated|
|11203686|         IYN (A)|2022-05-01 02:00:00|     0|               NV|        Not Validated|
|11203686|         IYN (A)|2022-05-01 01:30:00|     0|               NV|        Not Validated|
|11203686|         IBN (A)|2022-05-01 00:30:00|     0|               NV|        Not Validated|
|11203686|         IBN (A)|2022-05-01 01:30:00|     0|               NV|        Not Validated|
|11203686|         IBN (A)|2022-05-01 02:00:00|     0|               NV|        Not Validated|
|11203686|         IYN (A)|2022-05-01 01:00:00|     0|               NV|        Not Validated|
|11203686|         IYN (A)|2022-05-01 00:30:00|     0|               NV|        Not Validated|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 04:00:00|     0|              VAL|            validated|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 03:30:00|     0|              VAL|            validated|
|11203686|         IRN (A)|2022-05-01 02:00:00|     0|               NV|        Not Validated|
|11203686|         IRN (A)|2022-05-01 01:30:00|     0|               NV|        Not Validated|
|11203686|         IRN (A)|2022-05-01 01:00:00|     0|               NV|        Not Validated|
|11203686|         IRN (A)|2022-05-01 00:30:00|     0|               NV|        Not Validated|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 00:00:00|     0|              VAL|            validated|
+--------+----------------+-------------------+------+-----------------+---------------------+
```

Figure 3.0. Creating a new column for validation_status

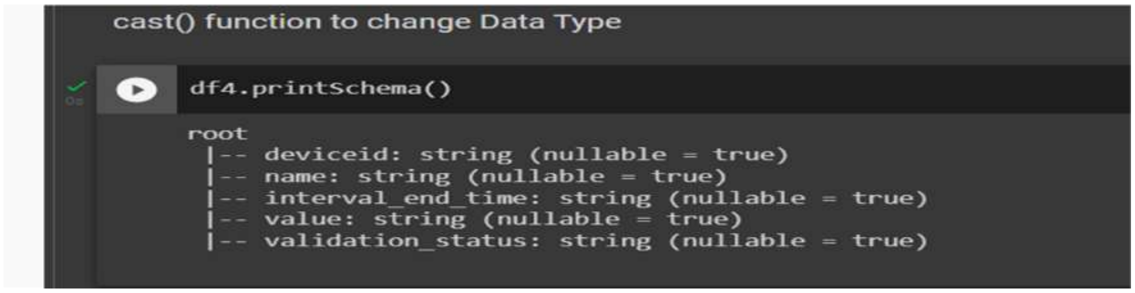


```
from pyspark.sql.functions import when
df4.withColumn("value_status", when(df4.value>=11.5,"greater ")
                               .when(df4.value<11.5,"lesser than 11.5")
                               .otherwise(df4.value)).show(30)
```

```
+--------+----------------+-------------------+------+-----------------+----------------+
|deviceid|            name|  interval_end_time| value|validation_status|    value_status|
+--------+----------------+-------------------+------+-----------------+----------------+
|11203686|         VYN (V)|2022-05-01 00:00:00| 251.1|               NV|        greater |
|11203686|         VYN (V)|2022-05-01 02:30:00|254.98|               NV|        greater |
|11203686|         VYN (V)|2022-05-01 03:00:00|253.58|               NV|        greater |
|11203686|         VYN (V)|2022-05-01 03:30:00|254.33|               NV|        greater |
|11203686|         VYN (V)|2022-05-01 04:00:00|254.75|               NV|        greater |
|11203686|         IYN (A)|2022-05-01 00:00:00|     0|               NV|lesser than 11.5|
|11203686|         IBN (A)|2022-05-01 00:00:00|     0|               NV|lesser than 11.5|
|11203686|         IBN (A)|2022-05-01 03:30:00|     0|               NV|lesser than 11.5|
|11203686|         IBN (A)|2022-05-01 03:00:00|     0|               NV|lesser than 11.5|
|11203686|         IYN (A)|2022-05-01 04:00:00|     0|               NV|lesser than 11.5|
|11203686|         IYN (A)|2022-05-01 03:30:00|     0|               NV|lesser than 11.5|
|11203686|         IYN (A)|2022-05-01 03:00:00|     0|               NV|lesser than 11.5|
|11203686|         IBN (A)|2022-05-01 02:30:00|     0|               NV|lesser than 11.5|
|11203686|         IYN (A)|2022-05-01 02:30:00|     0|               NV|lesser than 11.5|
|11203686|         IBN (A)|2022-05-01 04:00:00|     0|               NV|lesser than 11.5|
|11203686|         IBN (A)|2022-05-01 01:00:00|     0|               NV|lesser than 11.5|
|11203686|         IYN (A)|2022-05-01 02:00:00|     0|               NV|lesser than 11.5|
|11203686|         IYN (A)|2022-05-01 01:30:00|     0|               NV|lesser than 11.5|
|11203686|         IBN (A)|2022-05-01 00:30:00|     0|               NV|lesser than 11.5|
|11203686|         IBN (A)|2022-05-01 01:30:00|     0|               NV|lesser than 11.5|
|11203686|         IBN (A)|2022-05-01 02:00:00|     0|               NV|lesser than 11.5|
|11203686|         IYN (A)|2022-05-01 01:00:00|     0|               NV|lesser than 11.5|
|11203686|         IYN (A)|2022-05-01 00:30:00|     0|               NV|lesser than 11.5|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 04:00:00|     0|              VAL|lesser than 11.5|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 03:30:00|     0|              VAL|lesser than 11.5|
|11203686|         IRN (A)|2022-05-01 02:00:00|     0|               NV|lesser than 11.5|
|11203686|         IRN (A)|2022-05-01 01:30:00|     0|               NV|lesser than 11.5|
|11203686|         IRN (A)|2022-05-01 01:00:00|     0|               NV|lesser than 11.5|
|11203686|         IRN (A)|2022-05-01 00:30:00|     0|               NV|lesser than 11.5|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 00:00:00|     0|              VAL|lesser than 11.5|
```

Figure 3.1. Analysing the status of values on the basis given conditions

- **cast() Function:** In PySpark, you can cast or change the DataFrame column data type using cast() function of Column class. In the smart-meter dataset when we displayed the column type using the command printSchema(), we can see in figure 4.0 that all the column type have string data type.



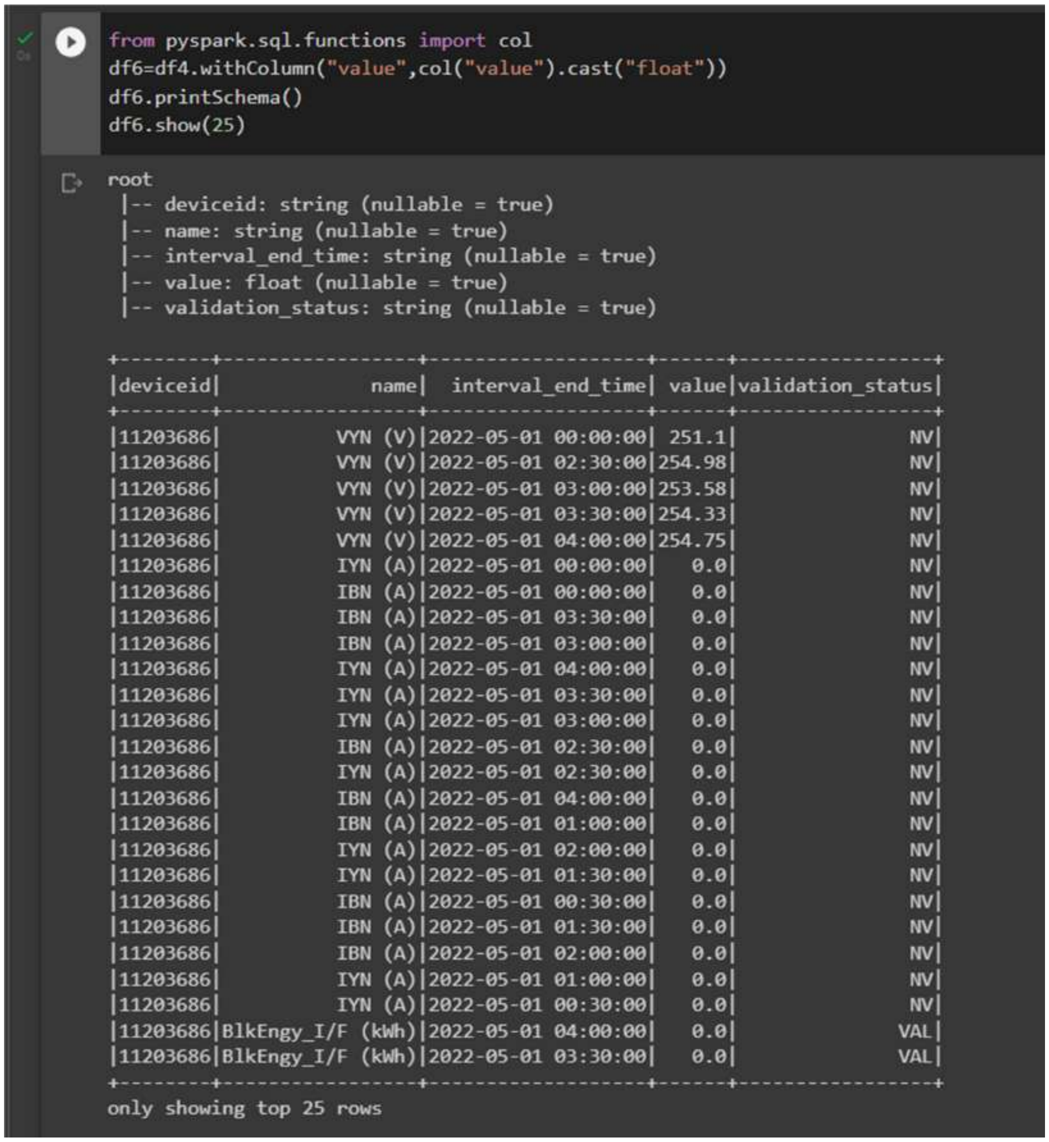
cast() function to change Data Type

```
df4.printSchema()
```

```
root
 |-- deviceid: string (nullable = true)
 |-- name: string (nullable = true)
 |-- interval_end_time: string (nullable = true)
 |-- value: string (nullable = true)
 |-- validation_status: string (nullable = true)
```

Figure 4.0. printSchema() of dataset "df4"

We created another data frame "df6" where the value column of each device is converted into float data type.



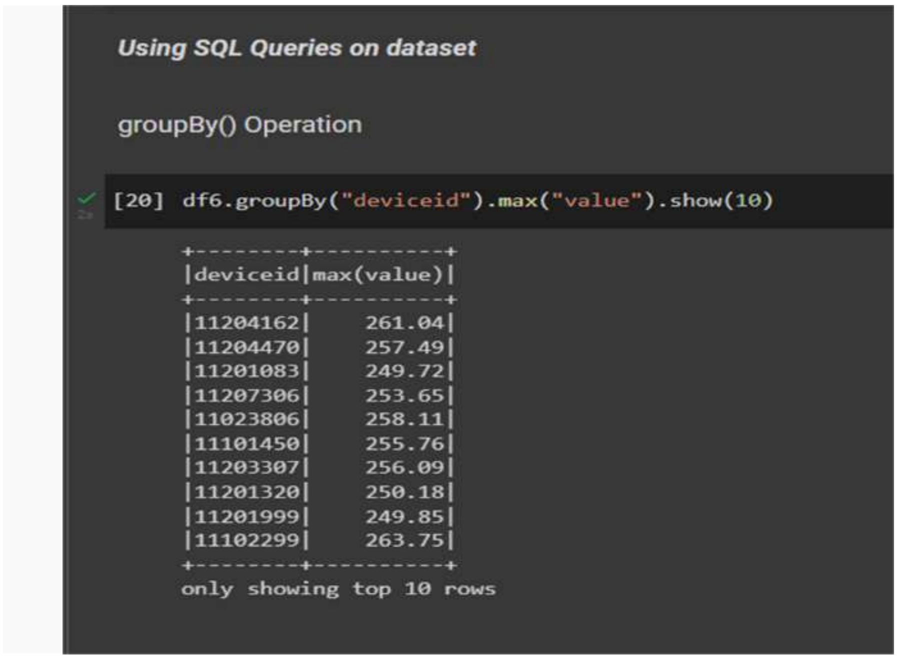
```
from pyspark.sql.functions import col
df6=df4.withColumn("value",col("value").cast("float"))
df6.printSchema()
df6.show(25)
```

```
root
 |-- deviceid: string (nullable = true)
 |-- name: string (nullable = true)
 |-- interval_end_time: string (nullable = true)
 |-- value: float (nullable = true)
 |-- validation_status: string (nullable = true)

+--------+-----------------+-------------------+------+-----------------+
|deviceid|             name|  interval_end_time| value|validation_status|
+--------+-----------------+-------------------+------+-----------------+
|11203686|          VYN (V)|2022-05-01 00:00:00| 251.1|               NV|
|11203686|          VYN (V)|2022-05-01 02:30:00|254.98|               NV|
|11203686|          VYN (V)|2022-05-01 03:00:00|253.58|               NV|
|11203686|          VYN (V)|2022-05-01 03:30:00|254.33|               NV|
|11203686|          VYN (V)|2022-05-01 04:00:00|254.75|               NV|
|11203686|          IYN (A)|2022-05-01 00:00:00|   0.0|               NV|
|11203686|          IBN (A)|2022-05-01 00:00:00|   0.0|               NV|
|11203686|          IBN (A)|2022-05-01 03:30:00|   0.0|               NV|
|11203686|          IBN (A)|2022-05-01 03:00:00|   0.0|               NV|
|11203686|          IYN (A)|2022-05-01 04:00:00|   0.0|               NV|
|11203686|          IYN (A)|2022-05-01 03:30:00|   0.0|               NV|
|11203686|          IYN (A)|2022-05-01 03:00:00|   0.0|               NV|
|11203686|          IBN (A)|2022-05-01 02:30:00|   0.0|               NV|
|11203686|          IYN (A)|2022-05-01 02:30:00|   0.0|               NV|
|11203686|          IBN (A)|2022-05-01 04:00:00|   0.0|               NV|
|11203686|          IBN (A)|2022-05-01 01:00:00|   0.0|               NV|
|11203686|          IYN (A)|2022-05-01 02:00:00|   0.0|               NV|
|11203686|          IYN (A)|2022-05-01 01:30:00|   0.0|               NV|
|11203686|          IBN (A)|2022-05-01 00:30:00|   0.0|               NV|
|11203686|          IBN (A)|2022-05-01 01:30:00|   0.0|               NV|
|11203686|          IBN (A)|2022-05-01 02:00:00|   0.0|               NV|
|11203686|          IYN (A)|2022-05-01 01:00:00|   0.0|               NV|
|11203686|          IYN (A)|2022-05-01 00:30:00|   0.0|               NV|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 04:00:00|   0.0|              VAL|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 03:30:00|   0.0|              VAL|
+--------+-----------------+-------------------+------+-----------------+
only showing top 25 rows
```

Figure 4.1. The value column's datatype changed to float datatype

- **SQL Commands, groupBy(), orderBy(), Window:** You can use either sort() or orderBy() function of PySpark DataFrame to sort DataFrame by ascending or descending order based on single or multiple columns, you can also do sorting using PySpark SQL sorting functions.
  Similar to SQL GROUP BY clause, PySpark groupBy() function is used to collect the identical data into groups on DataFrame and perform count, sum, avg, min, max functions on the grouped data.

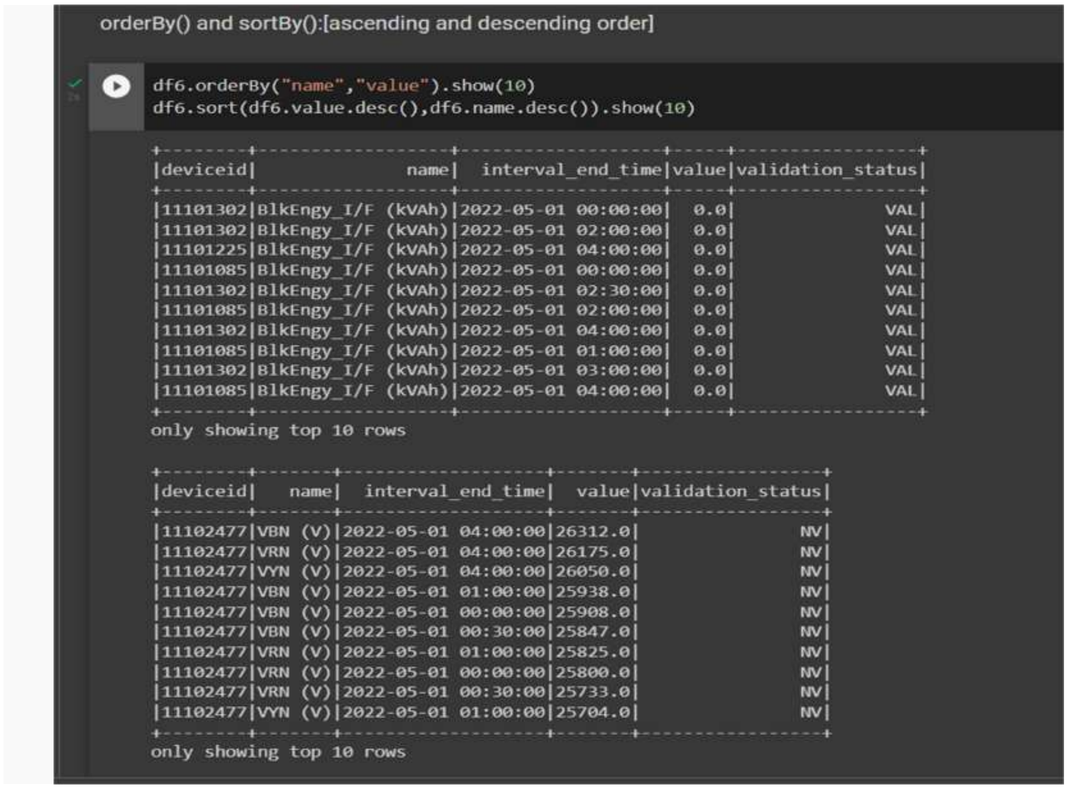
PySpark Window functions are used to calculate results such as the rank, row number etc. over a range of input rows. To perform an operation on a group first, we need to partition the data using Window.partitionBy(), and for row number and rank function we need to additionally order by on partition data using orderBy() clause.



```
[20] df6.groupBy("deviceid").max("value").show(10)

+--------+----------+
|deviceid|max(value)|
+--------+----------+
|11204162|    261.04|
|11204470|    257.49|
|11201083|    249.72|
|11207306|    253.65|
|11023806|    258.11|
|11101450|    255.76|
|11203307|    256.09|
|11201320|    250.18|
|11201999|    249.85|
|11102299|    263.75|
+--------+----------+
only showing top 10 rows
```

Figure 5.0.In the above dataset maximum value of each device is printed and is grouped by the ID of the devices

From the figure 5.0. we can infer we can simply see the maximum values of voltages for the devices corresponding to the ID of devices.

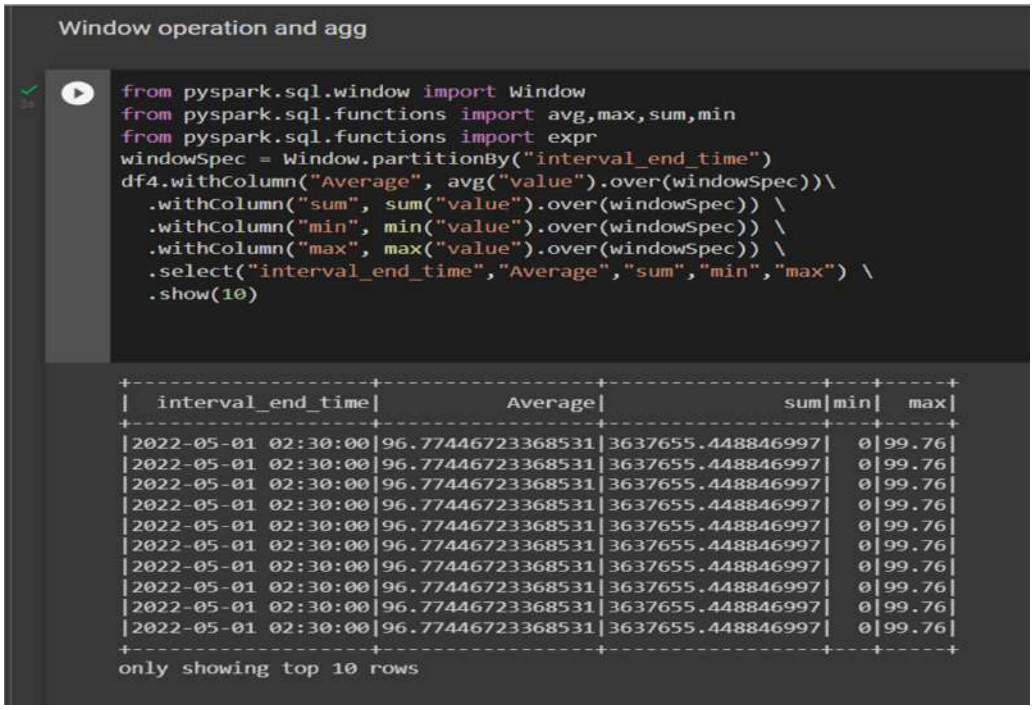


orderBy() and sortBy():[ascending and descending order]

```
df6.orderBy("name","value").show(10)
df6.sort(df6.value.desc(),df6.name.desc()).show(10)
```

```
+--------+-----------------+-------------------+-----+-----------------+
|deviceid|             name|  interval_end_time|value|validation_status|
+--------+-----------------+-------------------+-----+-----------------+
|11101302|BlkEngy_I/F (kVAh)|2022-05-01 00:00:00|  0.0|              VAL|
|11101302|BlkEngy_I/F (kVAh)|2022-05-01 02:00:00|  0.0|              VAL|
|11101225|BlkEngy_I/F (kVAh)|2022-05-01 04:00:00|  0.0|              VAL|
|11101085|BlkEngy_I/F (kVAh)|2022-05-01 00:00:00|  0.0|              VAL|
|11101302|BlkEngy_I/F (kVAh)|2022-05-01 02:30:00|  0.0|              VAL|
|11101085|BlkEngy_I/F (kVAh)|2022-05-01 02:00:00|  0.0|              VAL|
|11101302|BlkEngy_I/F (kVAh)|2022-05-01 04:00:00|  0.0|              VAL|
|11101085|BlkEngy_I/F (kVAh)|2022-05-01 01:00:00|  0.0|              VAL|
|11101302|BlkEngy_I/F (kVAh)|2022-05-01 03:00:00|  0.0|              VAL|
|11101085|BlkEngy_I/F (kVAh)|2022-05-01 04:00:00|  0.0|              VAL|
+--------+-----------------+-------------------+-----+-----------------+
only showing top 10 rows

+--------+-------+-------------------+-------+-----------------+
|deviceid|   name|  interval_end_time|  value|validation_status|
+--------+-------+-------------------+-------+-----------------+
|11102477|VBN (V)|2022-05-01 04:00:00|26312.0|               NV|
|11102477|VRN (V)|2022-05-01 04:00:00|26175.0|               NV|
|11102477|VYN (V)|2022-05-01 04:00:00|26050.0|               NV|
|11102477|VBN (V)|2022-05-01 01:00:00|25938.0|               NV|
|11102477|VBN (V)|2022-05-01 00:00:00|25908.0|               NV|
|11102477|VBN (V)|2022-05-01 00:30:00|25847.0|               NV|
|11102477|VRN (V)|2022-05-01 01:00:00|25825.0|               NV|
|11102477|VRN (V)|2022-05-01 00:00:00|25800.0|               NV|
|11102477|VRN (V)|2022-05-01 00:30:00|25733.0|               NV|
|11102477|VYN (V)|2022-05-01 01:00:00|25704.0|               NV|
+--------+-------+-------------------+-------+-----------------+
only showing top 10 rows
```

Figure 5.1.Using orderBy() and sortBy() to display data frame in order or sorting it in descending or ascending.



Window operation and agg

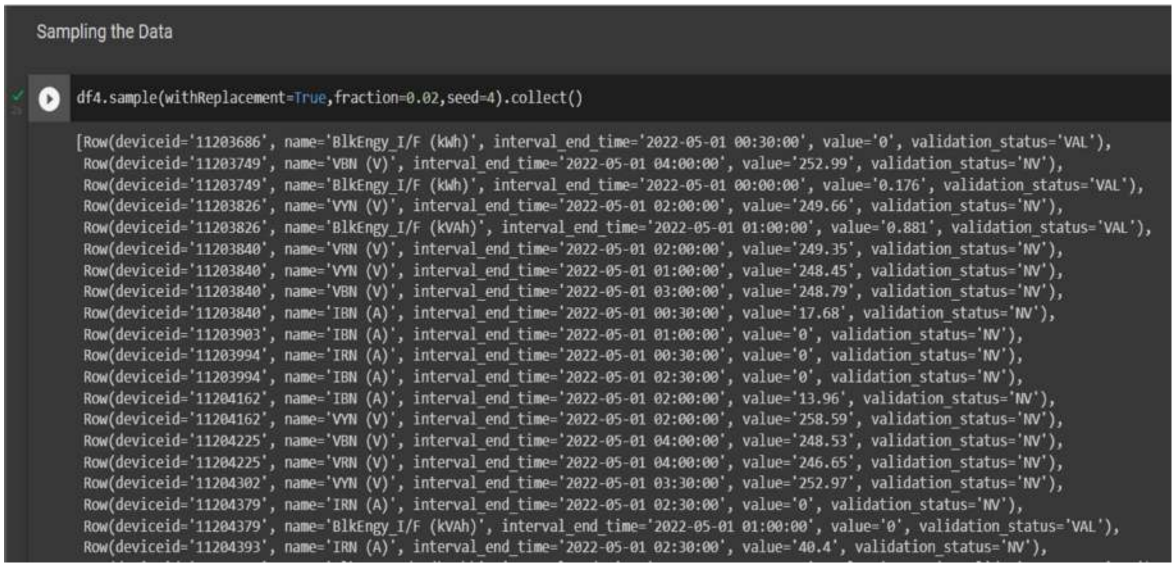
```
from pyspark.sql.window import Window
from pyspark.sql.functions import avg,max,sum,min
from pyspark.sql.functions import expr
windowSpec = Window.partitionBy("interval_end_time")
df4.withColumn("Average", avg("value").over(windowSpec))\
  .withColumn("sum", sum("value").over(windowSpec)) \
  .withColumn("min", min("value").over(windowSpec)) \
  .withColumn("max", max("value").over(windowSpec)) \
  .select("interval_end_time","Average","sum","min","max") \
  .show(10)
```

```
+-------------------+-----------------+-----------------+---+-----+
|  interval_end_time|          Average|              sum|min|  max|
+-------------------+-----------------+-----------------+---+-----+
|2022-05-01 02:30:00|96.77446723368531|3637655.448846997|  0|99.76|
|2022-05-01 02:30:00|96.77446723368531|3637655.448846997|  0|99.76|
|2022-05-01 02:30:00|96.77446723368531|3637655.448846997|  0|99.76|
|2022-05-01 02:30:00|96.77446723368531|3637655.448846997|  0|99.76|
|2022-05-01 02:30:00|96.77446723368531|3637655.448846997|  0|99.76|
|2022-05-01 02:30:00|96.77446723368531|3637655.448846997|  0|99.76|
|2022-05-01 02:30:00|96.77446723368531|3637655.448846997|  0|99.76|
|2022-05-01 02:30:00|96.77446723368531|3637655.448846997|  0|99.76|
|2022-05-01 02:30:00|96.77446723368531|3637655.448846997|  0|99.76|
|2022-05-01 02:30:00|96.77446723368531|3637655.448846997|  0|99.76|
+-------------------+-----------------+-----------------+---+-----+
only showing top 10 rows
```

Figure 5.2.Window functions with the aggregation operations like sum, avg, max, min

Window in Pyspark supports three kinds of function analytical, aggregation and ranking functions. In the above figure 5.2. Aggregative function is used to calculate the total count of voltage in dataset, sum of voltages in a particular interval of time, the maximum voltage of each interval, minimum and average of each interval end time.

- **Sampling of dataset:** Since the smart-meter dataset is very large and it is difficult to perform the analysis functions on the large datasets therefore many data analysts prefer to take sample of data set to analyse them. A small fraction according to one feasibility is taken and analysis is performed.
  PySpark sampling (pyspark.sql.dataframe.sample()) is a mechanism to get random sample records from the dataset, this is helpful when you have a larger dataset and wanted to analyse/test a subset of the data for example 2% of the original file.


Sampling the Data

```
df4.sample(withReplacement=True,fraction=0.02,seed=4).collect()
```

```
[Row(deviceid='11203686', name='BlkEngy_I/F (kWh)', interval_end_time='2022-05-01 00:30:00', value='0', validation_status='VAL'),
 Row(deviceid='11203749', name='VBN (V)', interval_end_time='2022-05-01 04:00:00', value='252.99', validation_status='NV'),
 Row(deviceid='11203749', name='BlkEngy_I/F (kWh)', interval_end_time='2022-05-01 00:00:00', value='0.176', validation_status='VAL'),
 Row(deviceid='11203826', name='VYN (V)', interval_end_time='2022-05-01 02:00:00', value='249.66', validation_status='NV'),
 Row(deviceid='11203826', name='BlkEngy_I/F (kVAh)', interval_end_time='2022-05-01 01:00:00', value='0.881', validation_status='VAL'),
 Row(deviceid='11203840', name='VRN (V)', interval_end_time='2022-05-01 02:00:00', value='249.35', validation_status='NV'),
 Row(deviceid='11203840', name='VYN (V)', interval_end_time='2022-05-01 01:00:00', value='248.45', validation_status='NV'),
 Row(deviceid='11203840', name='VBN (V)', interval_end_time='2022-05-01 03:00:00', value='248.79', validation_status='NV'),
 Row(deviceid='11203840', name='IBN (A)', interval_end_time='2022-05-01 00:30:00', value='17.68', validation_status='NV'),
 Row(deviceid='11203903', name='IBN (A)', interval_end_time='2022-05-01 01:00:00', value='0', validation_status='NV'),
 Row(deviceid='11203994', name='IRN (A)', interval_end_time='2022-05-01 00:30:00', value='0', validation_status='NV'),
 Row(deviceid='11203994', name='IBN (A)', interval_end_time='2022-05-01 02:30:00', value='0', validation_status='NV'),
 Row(deviceid='11204162', name='IBN (A)', interval_end_time='2022-05-01 02:00:00', value='13.96', validation_status='NV'),
 Row(deviceid='11204162', name='VYN (V)', interval_end_time='2022-05-01 02:00:00', value='258.59', validation_status='NV'),
 Row(deviceid='11204225', name='VBN (V)', interval_end_time='2022-05-01 04:00:00', value='248.53', validation_status='NV'),
 Row(deviceid='11204225', name='VRN (V)', interval_end_time='2022-05-01 04:00:00', value='246.65', validation_status='NV'),
 Row(deviceid='11204302', name='VYN (V)', interval_end_time='2022-05-01 03:30:00', value='252.97', validation_status='NV'),
 Row(deviceid='11204379', name='IRN (A)', interval_end_time='2022-05-01 02:30:00', value='0', validation_status='NV'),
 Row(deviceid='11204379', name='BlkEngy_I/F (kVAh)', interval_end_time='2022-05-01 01:00:00', value='0', validation_status='VAL'),
 Row(deviceid='11204393', name='IRN (A)', interval_end_time='2022-05-01 02:30:00', value='40.4', validation_status='NV'),
```

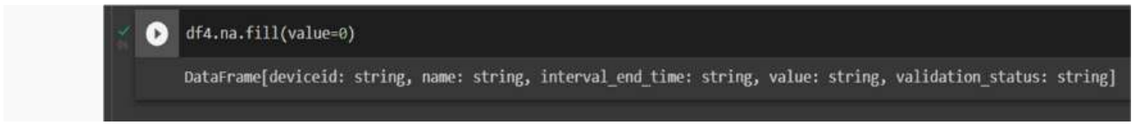
Figure 6.0. Sampling the dataset and collecting 2% of data with collect() command

Fraction parameter defines the subset or the part of dataset to be taken for analysis.

Seed parameter is used to reproduce the random sampling.

withReplacement parameter is used for collecting the replaceable or non replaceable sample.

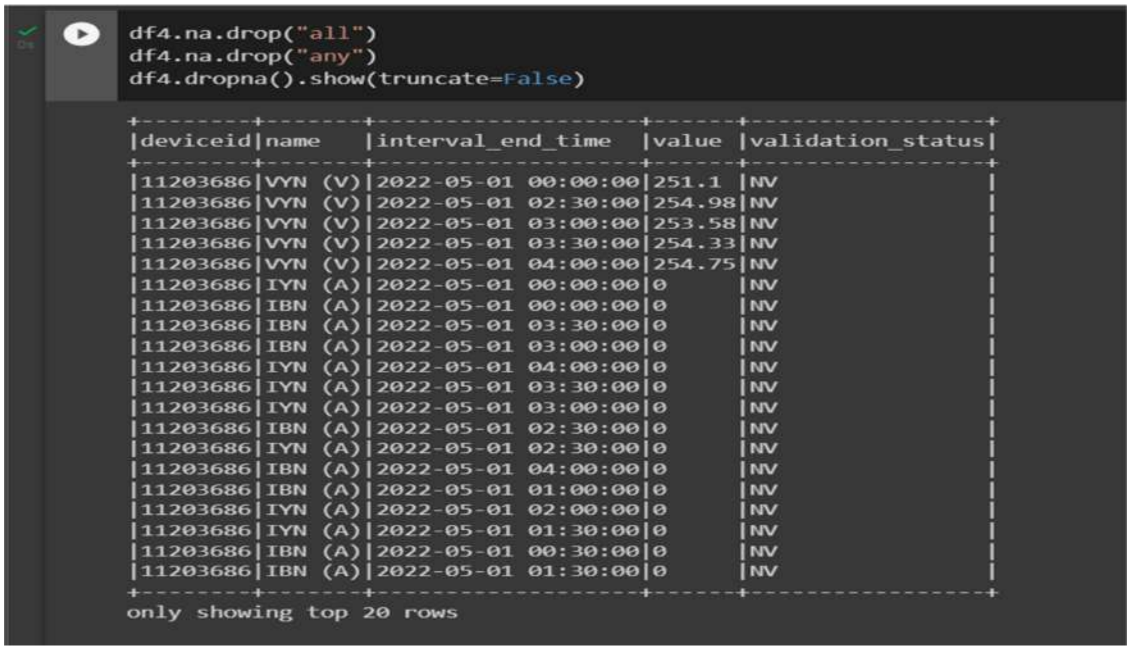
- **Handling missing values:** It is one of the most critical parts of any data exploration and analysis pipeline and when we have a large dataset so data engineers should have enough skills to handle the NA/missing values in the dataset. When we read the csv file of both the smart-meter datasets and when we displayed the description of each column in one of the dataset using printSchema() in figure 4.0. We saw that each column has "nullable=True" which means the columns corresponding have some missing or null values , therefore it is important to remove these null values for performing the data pre-processing.
  There are two ways for handling missing values: filling the null values with some other values and by dropping the null/missing values.



```
df4.na.fill(value=0)

DataFrame[deviceid: string, name: string, interval_end_time: string, value: string, validation_status: string]
```

Figure 7.0. Filling the null values with '0'

drop() function is used to remove/drop rows with NULL values in DataFrame columns, alternatively, you can also use df.na.drop().By using the drop() function you can drop all rows with null values in any, all, single, multiple, and selected columns. This function comes in handy when you need to clean the data before processing.

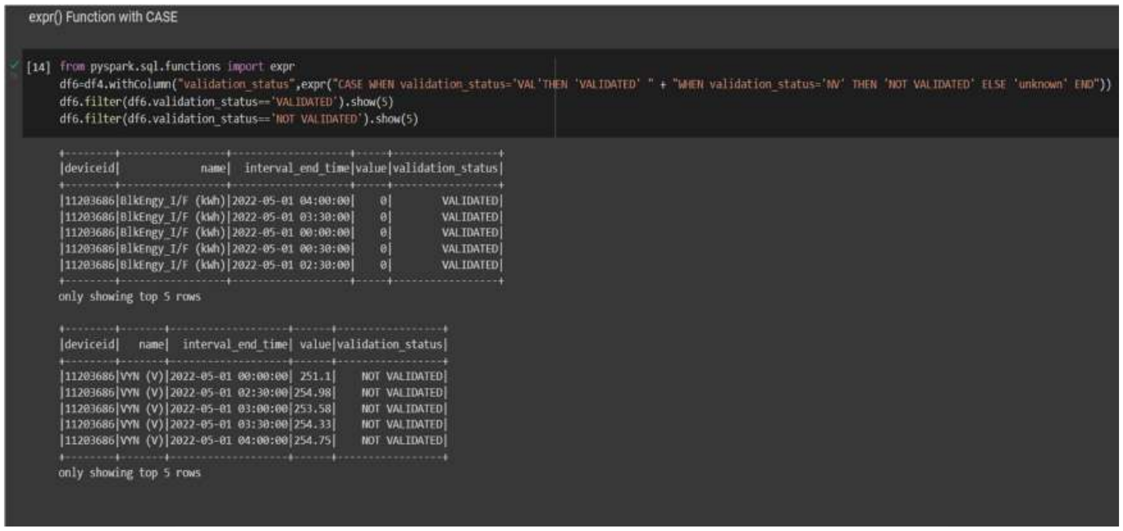


```
df4.na.drop("all")
df4.na.drop("any")
df4.dropna().show(truncate=False)
```

```
+--------+-------+-------------------+------+-----------------+
|deviceid|name   |interval_end_time  |value |validation_status|
+--------+-------+-------------------+------+-----------------+
|11203686|VYN (V)|2022-05-01 00:00:00|251.1 |NV               |
|11203686|VYN (V)|2022-05-01 02:30:00|254.98|NV               |
|11203686|VYN (V)|2022-05-01 03:00:00|253.58|NV               |
|11203686|VYN (V)|2022-05-01 03:30:00|254.33|NV               |
|11203686|VYN (V)|2022-05-01 04:00:00|254.75|NV               |
|11203686|IYN (A)|2022-05-01 00:00:00|0     |NV               |
|11203686|IBN (A)|2022-05-01 00:00:00|0     |NV               |
|11203686|IBN (A)|2022-05-01 03:30:00|0     |NV               |
|11203686|IBN (A)|2022-05-01 03:00:00|0     |NV               |
|11203686|IYN (A)|2022-05-01 04:00:00|0     |NV               |
|11203686|IYN (A)|2022-05-01 03:30:00|0     |NV               |
|11203686|IYN (A)|2022-05-01 03:00:00|0     |NV               |
|11203686|IBN (A)|2022-05-01 02:30:00|0     |NV               |
|11203686|IYN (A)|2022-05-01 02:30:00|0     |NV               |
|11203686|IBN (A)|2022-05-01 04:00:00|0     |NV               |
|11203686|IBN (A)|2022-05-01 01:00:00|0     |NV               |
|11203686|IYN (A)|2022-05-01 02:00:00|0     |NV               |
|11203686|IYN (A)|2022-05-01 01:30:00|0     |NV               |
|11203686|IBN (A)|2022-05-01 00:30:00|0     |NV               |
|11203686|IBN (A)|2022-05-01 01:30:00|0     |NV               |
+--------+-------+-------------------+------+-----------------+
only showing top 20 rows
```

Figure 7.1.Dropping null/missing values by using 3 commands yielding the same

- **expr() operation:** PySpark expr() is a SQL function to execute SQL-like expressions and to use an existing DataFrame column value as an expression argument to Pyspark built-in functions. Most of the commonly used SQL functions are either part of the PySpark column class or built-in pyspark.sql.functions API, besides these PySpark also supports many other SQL functions, so in order to use these, you have to use expr() function.



expr() Function with CASE

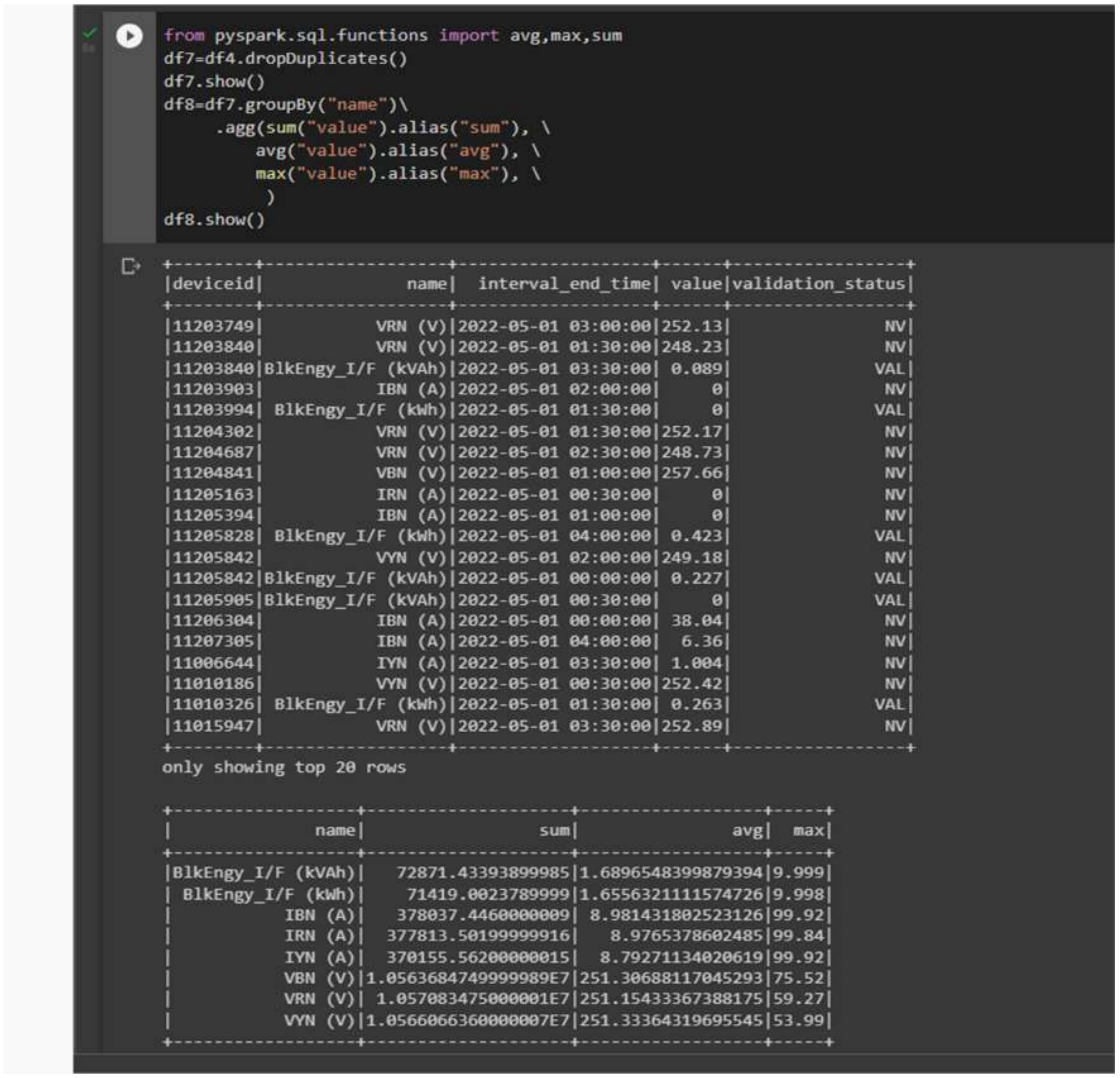
```
[14] from pyspark.sql.functions import expr
     df6=df4.withColumn("validation_status",expr("CASE WHEN validation_status='VAL'THEN 'VALIDATED' " + "WHEN validation_status='NV' THEN 'NOT VALIDATED' ELSE 'unknown' END"))
     df6.filter(df6.validation_status=='VALIDATED').show(5)
     df6.filter(df6.validation_status=='NOT VALIDATED').show(5)
```

```
+--------+----------------+-------------------+-----+-----------------+
|deviceid|            name|  interval_end_time|value|validation_status|
+--------+----------------+-------------------+-----+-----------------+
|11203686|BlkEngy_I/F (kWh)|2022-05-01 04:00:00|    0|        VALIDATED|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 03:30:00|    0|        VALIDATED|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 00:00:00|    0|        VALIDATED|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 00:30:00|    0|        VALIDATED|
|11203686|BlkEngy_I/F (kWh)|2022-05-01 02:30:00|    0|        VALIDATED|
+--------+----------------+-------------------+-----+-----------------+
only showing top 5 rows

+--------+-------+-------------------+------+-----------------+
|deviceid|   name|  interval_end_time| value|validation_status|
+--------+-------+-------------------+------+-----------------+
|11203686|VYN (V)|2022-05-01 00:00:00| 251.1|    NOT VALIDATED|
|11203686|VYN (V)|2022-05-01 02:30:00|254.98|    NOT VALIDATED|
|11203686|VYN (V)|2022-05-01 03:00:00|253.58|    NOT VALIDATED|
|11203686|VYN (V)|2022-05-01 03:30:00|254.33|    NOT VALIDATED|
|11203686|VYN (V)|2022-05-01 04:00:00|254.75|    NOT VALIDATED|
+--------+-------+-------------------+------+-----------------+
only showing top 5 rows
```

Figure 8.0. expr() function used with the CASE and WHEN

Here in the figure 8.0 we had to first import the sql function expr() and the changes were made in the column “validation_status” and using expr() function the devices having validation_status as “NV” is converted into “NOT VALIDATED” and similarly devices having “VAL” is converted to “VALIDATED”.
In the example shown filter() command is also given to filter out the “VALIDATED” devices separately.

- **dropDuplicates():** this command is given to drop duplicate rows.

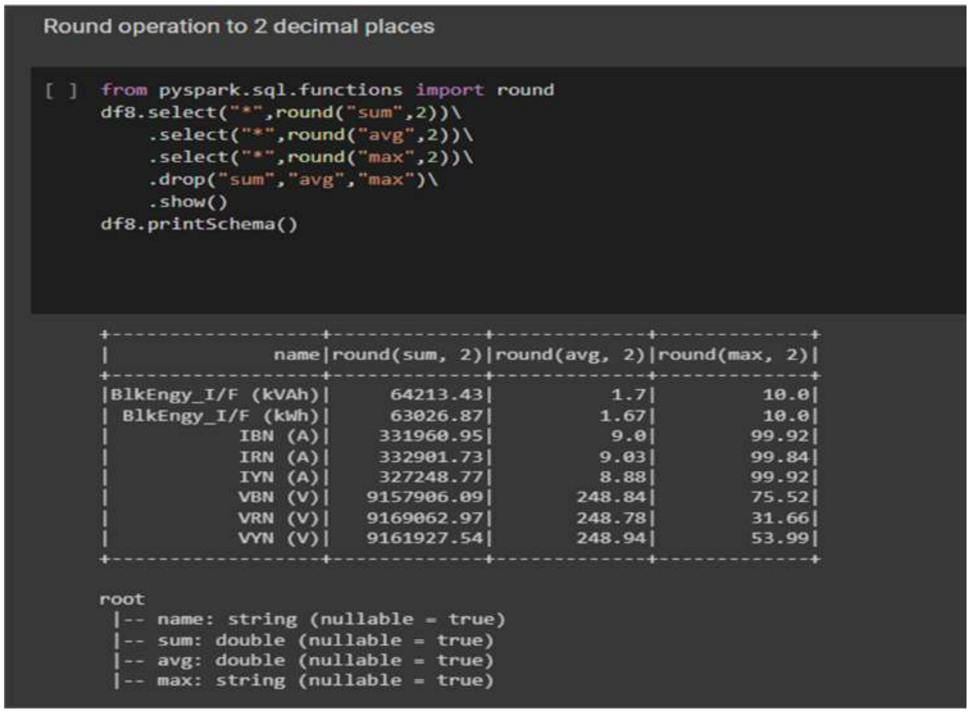


```
from pyspark.sql.functions import avg,max,sum
df7=df4.dropDuplicates()
df7.show()
df8=df7.groupBy("name")\
     .agg(sum("value").alias("sum"), \
         avg("value").alias("avg"), \
         max("value").alias("max"), \
          )
df8.show()
```

```
+--------+-----------------+-------------------+------+-----------------+
|deviceid|             name|  interval_end_time| value|validation_status|
+--------+-----------------+-------------------+------+-----------------+
|11203749|          VRN (V)|2022-05-01 03:00:00|252.13|               NV|
|11203840|          VRN (V)|2022-05-01 01:30:00|248.23|               NV|
|11203840|BlkEngy_I/F (kVAh)|2022-05-01 03:30:00| 0.089|              VAL|
|11203903|          IBN (A)|2022-05-01 02:00:00|     0|               NV|
|11203994| BlkEngy_I/F (kWh)|2022-05-01 01:30:00|     0|              VAL|
|11204302|          VRN (V)|2022-05-01 01:30:00|252.17|               NV|
|11204687|          VRN (V)|2022-05-01 02:30:00|248.73|               NV|
|11204841|          VBN (V)|2022-05-01 01:00:00|257.66|               NV|
|11205163|          IRN (A)|2022-05-01 00:30:00|     0|               NV|
|11205394|          IBN (A)|2022-05-01 01:00:00|     0|               NV|
|11205828| BlkEngy_I/F (kWh)|2022-05-01 04:00:00| 0.423|              VAL|
|11205842|          VYN (V)|2022-05-01 02:00:00|249.18|               NV|
|11205842|BlkEngy_I/F (kVAh)|2022-05-01 00:00:00| 0.227|              VAL|
|11205905|BlkEngy_I/F (kVAh)|2022-05-01 00:30:00|     0|              VAL|
|11206304|          IBN (A)|2022-05-01 00:00:00| 38.04|               NV|
|11207305|          IBN (A)|2022-05-01 04:00:00|  6.36|               NV|
|11006644|          IYN (A)|2022-05-01 03:30:00| 1.004|               NV|
|11010186|          VYN (V)|2022-05-01 00:30:00|252.42|               NV|
|11010326| BlkEngy_I/F (kWh)|2022-05-01 01:30:00| 0.263|              VAL|
|11015947|          VRN (V)|2022-05-01 03:30:00|252.89|               NV|
+--------+-----------------+-------------------+------+-----------------+
only showing top 20 rows

+------------------+--------------------+------------------+-----+
|              name|                 sum|               avg|  max|
+------------------+--------------------+------------------+-----+
|BlkEngy_I/F (kVAh)|   72871.43393899985|1.6896548399879394|9.999|
| BlkEngy_I/F (kWh)|   71419.0023789999|1.6556321111574726|9.998|
|           IBN (A)|   378037.4460000009| 8.981431802523126|99.92|
|           IRN (A)|  377813.50199999916|   8.9765378602485|99.84|
|           IYN (A)|  370155.56200000015|  8.79271134020619|99.92|
|           VBN (V)|1.0563684749999989E7|251.30688117045293|75.52|
|           VRN (V)| 1.057083475000001E7|251.15433367388175|59.27|
|           VYN (V)|1.0566066360000007E7|251.33364319695545|53.99|
+------------------+--------------------+------------------+-----+
```

Figure 9.0. grouping the dataset by “name” then applying the aggregative functions and created a new dataset for sum, avg, max values.

The values in the new dataset created after dropping the duplicate rows, the values are very large . Hence we round the values to any number of decimal places (say 2) we can change the decimal point.



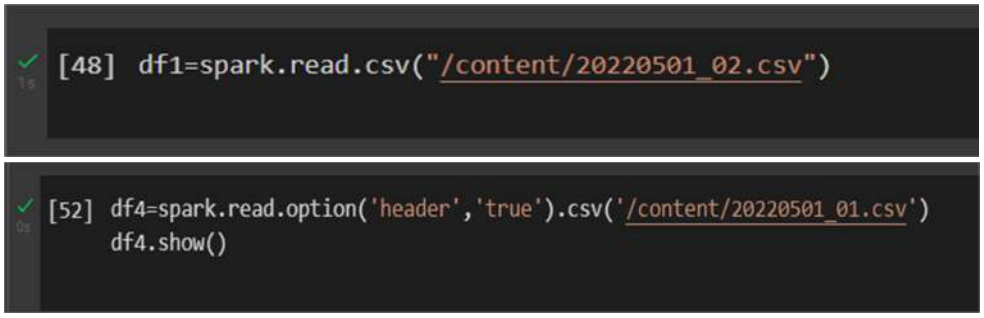
Round operation to 2 decimal places

```
[ ] from pyspark.sql.functions import round
    df8.select("*",round("sum",2))\
        .select("*",round("avg",2))\
        .select("*",round("max",2))\
        .drop("sum","avg","max")\
        .show()
    df8.printSchema()
```

```
+------------------+-------------+-------------+-------------+
|              name|round(sum, 2)|round(avg, 2)|round(max, 2)|
+------------------+-------------+-------------+-------------+
|BlkEngy_I/F (kVAh)|     64213.43|          1.7|         10.0|
| BlkEngy_I/F (kWh)|     63026.87|         1.67|         10.0|
|           IBN (A)|    331960.95|          9.0|        99.92|
|           IRN (A)|    332901.73|         9.03|        99.84|
|           IYN (A)|    327248.77|         8.88|        99.92|
|           VBN (V)|   9157906.09|       248.84|        75.52|
|           VRN (V)|   9169062.97|       248.78|        31.66|
|           VYN (V)|   9161927.54|       248.94|        53.99|
+------------------+-------------+-------------+-------------+

root
 |-- name: string (nullable = true)
 |-- sum: double (nullable = true)
 |-- avg: double (nullable = true)
 |-- max: string (nullable = true)
```

Figure 9.1.Rounding voltage values up to 2 decimal places

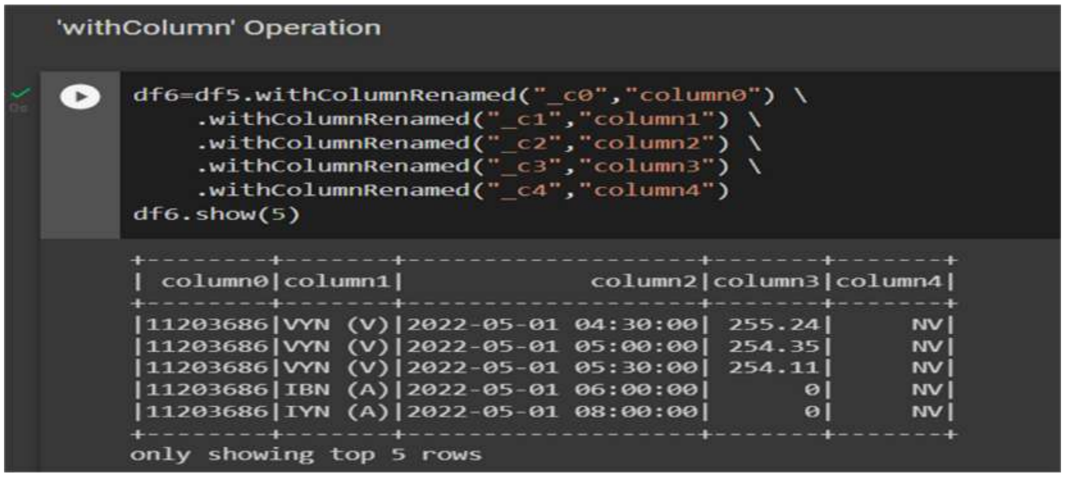
- **join(), split(), Union() commands:** PySpark Join is used to combine two Data Frames and by chaining these you can join multiple Data Frames; it supports all basic join type operations available in traditional SQL like inner, left outer, right outer, cross, self, join().
  First we read the 2 datasets :



```
[48] df1=spark.read.csv("/content/20220501_02.csv")
```

```
[52] df4=spark.read.option('header','true').csv('/content/20220501_01.csv')
     df4.show()
```

Then with the help of withColumn() command we changed the names of the column, for df1.

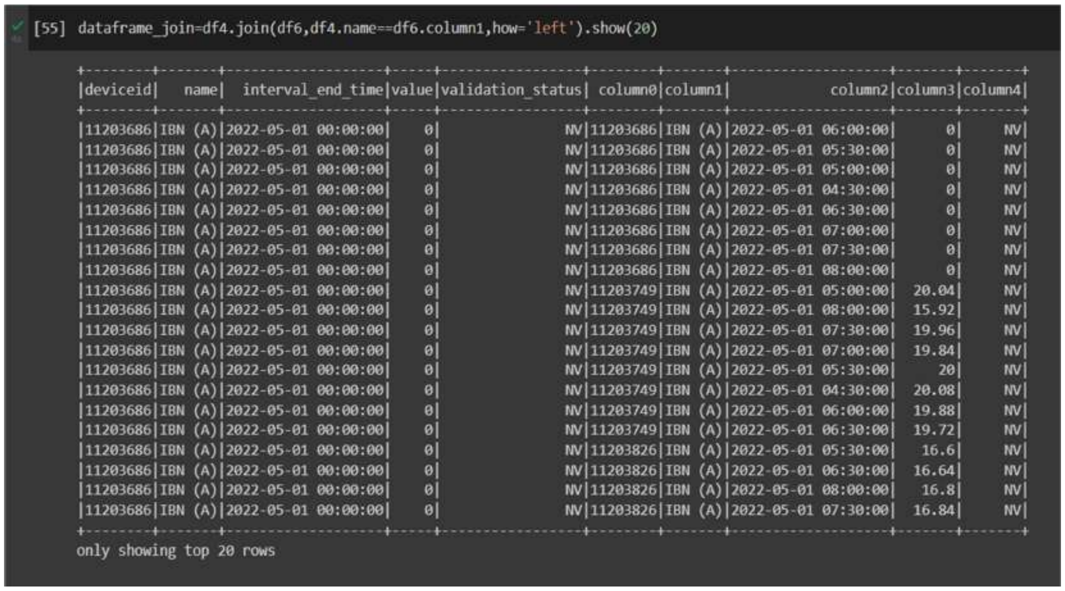


'withColumn' Operation

```
df6=df5.withColumnRenamed("_c0","column0") \
    .withColumnRenamed("_c1","column1") \
    .withColumnRenamed("_c2","column2") \
    .withColumnRenamed("_c3","column3") \
    .withColumnRenamed("_c4","column4")
df6.show(5)
```

```
+--------+-------+-------------------+-------+-------+
| column0|column1|            column2|column3|column4|
+--------+-------+-------------------+-------+-------+
|11203686|VYN (V)|2022-05-01 04:30:00| 255.24|     NV|
|11203686|VYN (V)|2022-05-01 05:00:00| 254.35|     NV|
|11203686|VYN (V)|2022-05-01 05:30:00| 254.11|     NV|
|11203686|IBN (A)|2022-05-01 06:00:00|      0|     NV|
|11203686|IYN (A)|2022-05-01 08:00:00|      0|     NV|
+--------+-------+-------------------+-------+-------+
only showing top 5 rows
```

Then using join() command we merged the 2 datasets to analyse both the datasets or to perform the operation on both the datasets simultaneously.



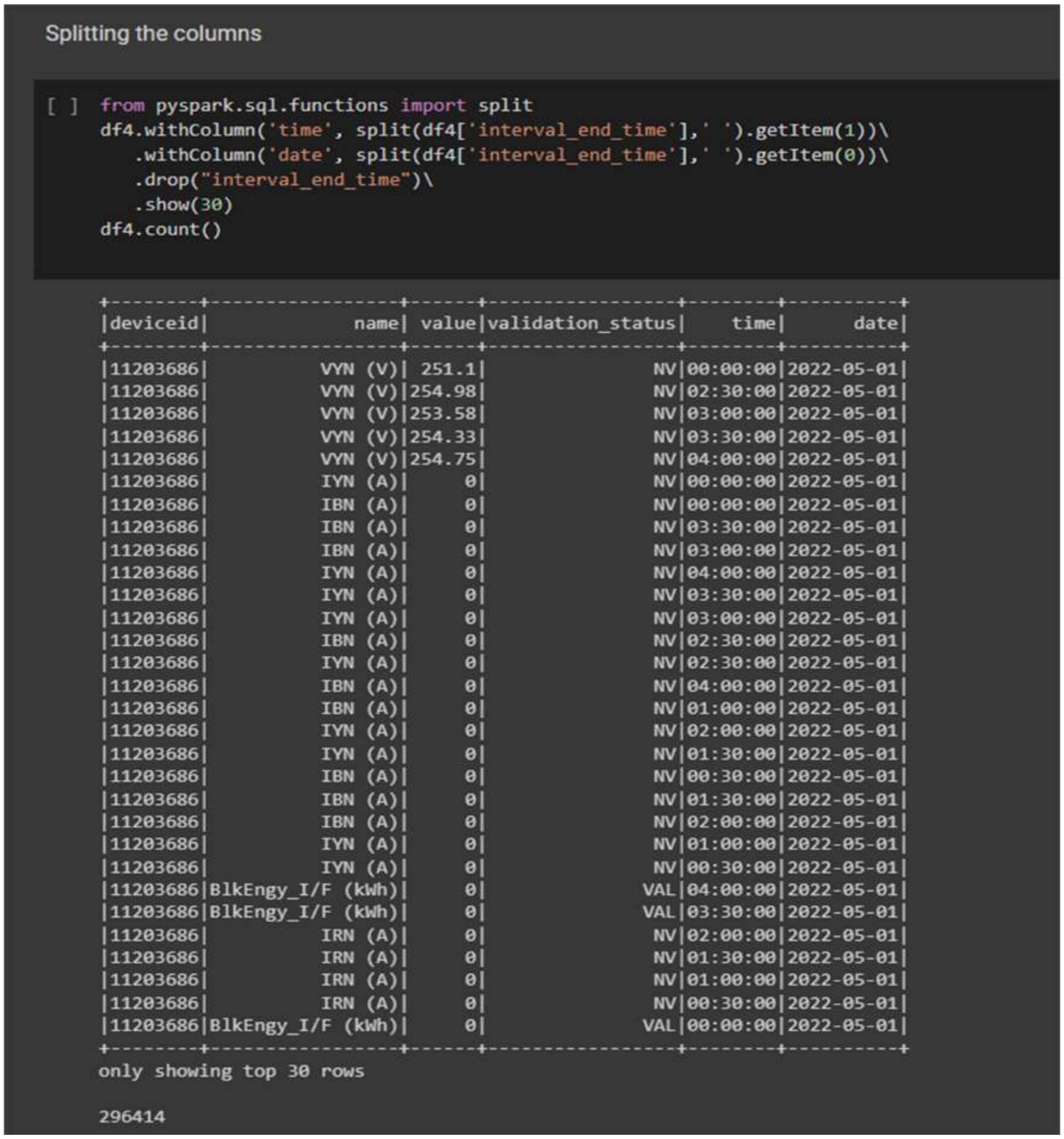
```
[55] dataframe_join=df4.join(df6,df4.name==df6.column1,how='left').show(20)
```

```
+--------+-------+-------------------+-----+-----------------+--------+-------+-------------------+-------+-------+
|deviceid|   name|  interval_end_time|value|validation_status| column0|column1|            column2|column3|column4|
+--------+-------+-------------------+-----+-----------------+--------+-------+-------------------+-------+-------+
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203686|IBN (A)|2022-05-01 06:00:00|      0|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203686|IBN (A)|2022-05-01 05:30:00|      0|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203686|IBN (A)|2022-05-01 05:00:00|      0|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203686|IBN (A)|2022-05-01 04:30:00|      0|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203686|IBN (A)|2022-05-01 06:30:00|      0|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203686|IBN (A)|2022-05-01 07:00:00|      0|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203686|IBN (A)|2022-05-01 07:30:00|      0|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203686|IBN (A)|2022-05-01 08:00:00|      0|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203749|IBN (A)|2022-05-01 05:00:00|  20.04|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203749|IBN (A)|2022-05-01 08:00:00|  15.92|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203749|IBN (A)|2022-05-01 07:30:00|  19.96|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203749|IBN (A)|2022-05-01 07:00:00|  19.84|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203749|IBN (A)|2022-05-01 05:30:00|     20|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203749|IBN (A)|2022-05-01 04:30:00|  20.08|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203749|IBN (A)|2022-05-01 06:00:00|  19.88|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203749|IBN (A)|2022-05-01 06:30:00|  19.72|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203826|IBN (A)|2022-05-01 05:30:00|   16.6|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203826|IBN (A)|2022-05-01 06:30:00|  16.64|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203826|IBN (A)|2022-05-01 08:00:00|   16.8|     NV|
|11203686|IBN (A)|2022-05-01 00:00:00|    0|               NV|11203826|IBN (A)|2022-05-01 07:30:00|  16.84|     NV|
+--------+-------+-------------------+-----+-----------------+--------+-------+-------------------+-------+-------+
only showing top 20 rows
```

While creating a new dataframe after joining we used the condition for joining that is the name column of df4 dataset should be equal to the name column of df6 that is ‘column1’. The how option specifies how the data frames should be joined, whether from left inner join etc.

In the above example the dataframe df4 is joined to the left of the data frame df6.

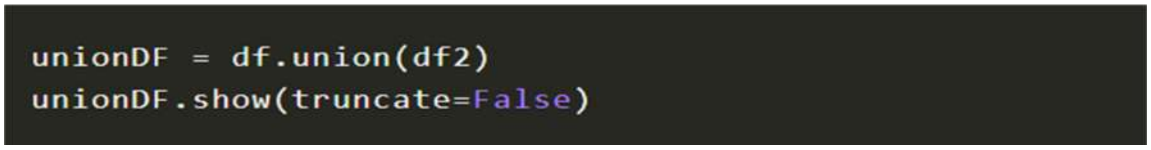
For splitting the columns into 2 or three we use the split() command. For example in the smart meter dataset-2 the column ‘interval_end_time’ has values separated by ‘-’ and “ ” therefore it can be splitted into 2 columns as date and time. And to get the number of rows we displayed using count().


Splitting the columns

```
[ ] from pyspark.sql.functions import split
    df4.withColumn('time', split(df4['interval_end_time'],' ').getItem(1))\
       .withColumn('date', split(df4['interval_end_time'],' ').getItem(0))\
       .drop("interval_end_time")\
       .show(30)
    df4.count()
```

```
+--------+-----------------+------+-----------------+--------+----------+
|deviceid|             name| value|validation_status|    time|      date|
+--------+-----------------+------+-----------------+--------+----------+
|11203686|          VYN (V)| 251.1|               NV|00:00:00|2022-05-01|
|11203686|          VYN (V)|254.98|               NV|02:30:00|2022-05-01|
|11203686|          VYN (V)|253.58|               NV|03:00:00|2022-05-01|
|11203686|          VYN (V)|254.33|               NV|03:30:00|2022-05-01|
|11203686|          VYN (V)|254.75|               NV|04:00:00|2022-05-01|
|11203686|          IYN (A)|     0|               NV|00:00:00|2022-05-01|
|11203686|          IBN (A)|     0|               NV|00:00:00|2022-05-01|
|11203686|          IBN (A)|     0|               NV|03:30:00|2022-05-01|
|11203686|          IBN (A)|     0|               NV|03:00:00|2022-05-01|
|11203686|          IYN (A)|     0|               NV|04:00:00|2022-05-01|
|11203686|          IYN (A)|     0|               NV|03:30:00|2022-05-01|
|11203686|          IYN (A)|     0|               NV|03:00:00|2022-05-01|
|11203686|          IBN (A)|     0|               NV|02:30:00|2022-05-01|
|11203686|          IYN (A)|     0|               NV|02:30:00|2022-05-01|
|11203686|          IBN (A)|     0|               NV|04:00:00|2022-05-01|
|11203686|          IBN (A)|     0|               NV|01:00:00|2022-05-01|
|11203686|          IYN (A)|     0|               NV|02:00:00|2022-05-01|
|11203686|          IYN (A)|     0|               NV|01:30:00|2022-05-01|
|11203686|          IBN (A)|     0|               NV|00:30:00|2022-05-01|
|11203686|          IBN (A)|     0|               NV|01:30:00|2022-05-01|
|11203686|          IBN (A)|     0|               NV|02:00:00|2022-05-01|
|11203686|          IYN (A)|     0|               NV|01:00:00|2022-05-01|
|11203686|          IYN (A)|     0|               NV|00:30:00|2022-05-01|
|11203686|BlkEngy_I/F (kWh)|     0|              VAL|04:00:00|2022-05-01|
|11203686|BlkEngy_I/F (kWh)|     0|              VAL|03:30:00|2022-05-01|
|11203686|          IRN (A)|     0|               NV|02:00:00|2022-05-01|
|11203686|          IRN (A)|     0|               NV|01:30:00|2022-05-01|
|11203686|          IRN (A)|     0|               NV|01:00:00|2022-05-01|
|11203686|          IRN (A)|     0|               NV|00:30:00|2022-05-01|
|11203686|BlkEngy_I/F (kWh)|     0|              VAL|00:00:00|2022-05-01|
+--------+-----------------+------+-----------------+--------+----------+
only showing top 30 rows

296414
```

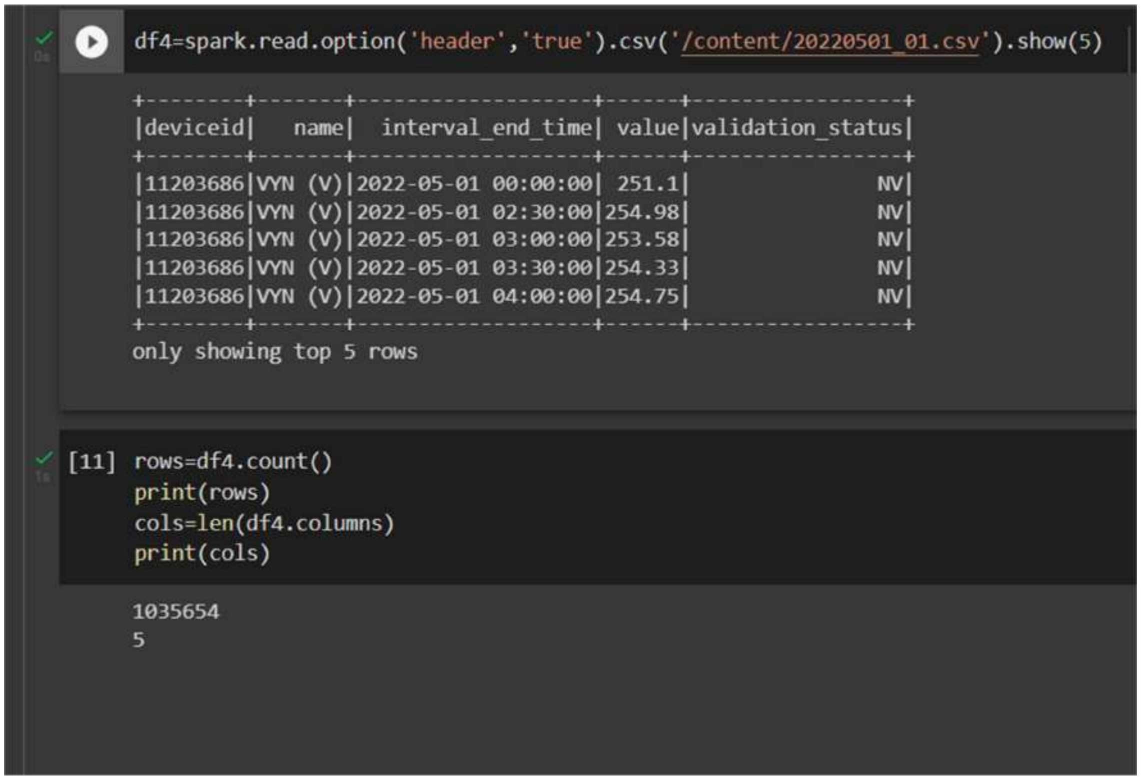
DataFrame union() method merges two data frames and returns the new DataFrame with all rows from two data frames regardless of duplicate data.


```
unionDF = df.union(df2)
unionDF.show(truncate=False)
```

Section 5.0. ROW and COLUMN functions

PYSPARK ROW is a class that represents the Data Frame as a record. We can create row objects in PySpark by certain parameters in PySpark. The row class extends the tuple, so the variable arguments are open while creating the row class. We can create a row object and can retrieve the data from the Row In PySpark, the withColumn() function is widely used and defined as the transformation function of the DataFrame which is further used to change the value, convert the datatype of an existing column, create the new column etc.

We can count the number of rows and columns in a dataset. For displaying the number of rows and columns both has different commands. For row count we use count() and for column count we use len()

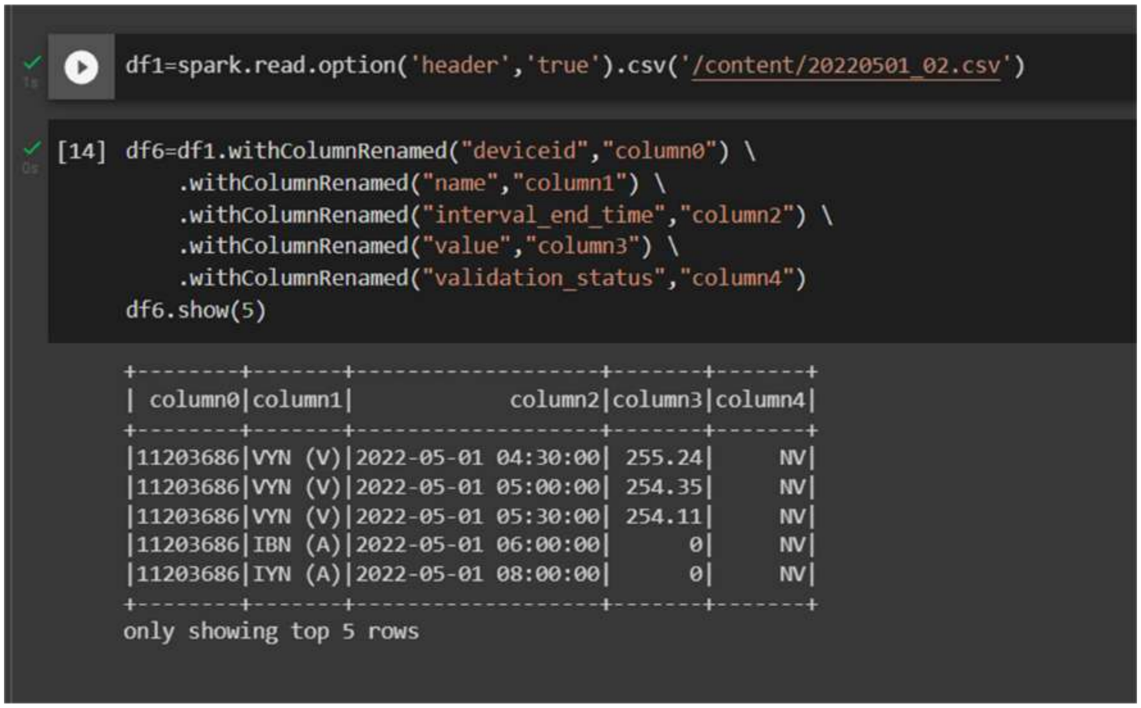


```
df4=spark.read.option('header','true').csv('/content/20220501_01.csv').show(5)

+--------+-------+-------------------+------+-----------------+
|deviceid|   name|  interval_end_time| value|validation_status|
+--------+-------+-------------------+------+-----------------+
|11203686|VYN (V)|2022-05-01 00:00:00| 251.1|               NV|
|11203686|VYN (V)|2022-05-01 02:30:00|254.98|               NV|
|11203686|VYN (V)|2022-05-01 03:00:00|253.58|               NV|
|11203686|VYN (V)|2022-05-01 03:30:00|254.33|               NV|
|11203686|VYN (V)|2022-05-01 04:00:00|254.75|               NV|
+--------+-------+-------------------+------+-----------------+
only showing top 5 rows
```

```
[11] rows=df4.count()
     print(rows)
     cols=len(df4.columns)
     print(cols)

     1035654
     5
```

Apart from counting the number of rows and columns we can split a column into 2 or more as we have seen by using split() command we can also rename a column:

By using withColumnRenamed() or alias()



```
df1=spark.read.option('header','true').csv('/content/20220501_02.csv')
```

```
[14] df6=df1.withColumnRenamed("deviceid","column0") \
         .withColumnRenamed("name","column1") \
         .withColumnRenamed("interval_end_time","column2") \
         .withColumnRenamed("value","column3") \
         .withColumnRenamed("validation_status","column4")
     df6.show(5)
```

```
+--------+-------+-------------------+-------+-------+
| column0|column1|            column2|column3|column4|
+--------+-------+-------------------+-------+-------+
|11203686|VYN (V)|2022-05-01 04:30:00| 255.24|     NV|
|11203686|VYN (V)|2022-05-01 05:00:00| 254.35|     NV|
|11203686|VYN (V)|2022-05-01 05:30:00| 254.11|     NV|
|11203686|IBN (A)|2022-05-01 06:00:00|      0|     NV|
|11203686|IYN (A)|2022-05-01 08:00:00|      0|     NV|
+--------+-------+-------------------+-------+-------+
only showing top 5 rows
```

The df6 data frame is created same as df1 but the column names are renamed according to the users or viewers convenience.

We can select any column to display any particular data rather than displaying whole data frame using select() command.

Section 6.0. Reviewing results from Analysis

The results from the PySpark can be reviewed, we used may commands and functions of pyspark on the dataset of smart-meter. First we loaded the data set in CSV file format. The CSV file format was converted to parquet format since parquet form since parquet files maintains the schema along with the data and is used to process a structured file. After loading and writing the data into a specific file format we performed filter() command along with select(). We can filter out the values and conditions can also be used in filter command such as ==, <=, >=, != on selected columns that are needed to be changed or analysed. SQL commands such as groupBy(), orderBy() are used to sort the voltage values in the data set in ascending or descending order if we have to sort the values we can also use sort() command. The devices that are validated are displayed separately by using expr() function or it can be done with help ‘when’ and ‘otherwise’ operation. If we have to analyse devices with high or low voltage value then again we can use ‘CASE’ along with expr(). To display the number of records in a dataset we can use count() or len(). Window operation is used to operate on group of row such as we can perform ranking analysis, aggregative analysis or analytic functions on the rows having same elements according to which they are partition By. The sum and average values for each device are calculated and the maximum and minimum voltage values are displayed corresponding to their device id. Since the proactive voltage analysis is done on a very large datasets so we perform sampling of the data by taking a particular fraction of the original file and on a fraction of dataframe pre-processing is performed and then applied to whole datasets. We also perform partitioning of datasets that are large. The partitioning of dataset is done randomly to split dataset into training and test dataset so that processing can be performed on training data and to execute and validate it can be applied on the other fraction that is the test dataset. The splitting of the columns help to split the time intervals into separate columns of date and time or months, days and years. With the help of pyspark we can also handle the null/missing values in the datasets either we can drop null/missing values or we can fill the null values with the specific element or number. During data pre-processing of such large smart-meter data we encounter many null/missing values so pyspark helps making this easier by giving the commands drop.na() or fill.na().

Section 7.0. Summary of the Project

In this project we have seen how smart meter works and is helpful for the proactive voltage analysis. Pyspark is useful for carrying out the data pre-processing in the analysis of smart meter dataset given. The working of smart-meter and how it performs a two way communication between the customers and the source was discussed. Pyspark helps to study the low and high voltages present in the data to perform SQL functions, to partition the data into smaller datasets to ease the visualization. The structure and architecture of pyspark was discussed representing how it works as a mater slave way. The pyspark having entry points and SparkSession or SparkContext both perform as an entry points to the application for performing the actions and transformation operations. Then we encountered the modules and packages of the pyspark but since our project for voltage analysis is based on dataframe analysis hence we only focused on Pyspark Dataframe, SQL functions and RDD operations. We performed the SQL queries of pyspark on the smart meter dataset after loading the csv file format, converted the csv file into parquet format. We studied the dataframe operations that can be performed the analysis, and reviewed the results. The code snippets showing how each command outputs the data is also shown for the understanding of the pyspark dataframe operations. Handling of missing values and sampling of the datasets was also studied and outputs showing the results summarize how pyspark can be useful for converting large datasets into a sample/fraction of the original file and handle the null/missing values without interrupting the non-null values. As the Smart Districts will have smart meters at consumer end. Therefore for pro-active observation of voltage profile of the consumers, we can utilize the smart meter data for this purpose.

Commands/Functions Syntax	**Purpose/ Functionality performed**
!pip install pyspark	This command installs the necessary packages py4j and pyspark and successfully built the pyspark.
from pyspark.sql import SparkSession or spark.sparkContext	Both acts as an entry point to the pyspark but sparksession replaced sparkcontext, not completely.
spark.read.csv("path")	Reads the csv file
spark.createDataFrame("")	Creates a data frame from the users input
dataframe.write.parquet("path")	This command will create any file format into parquet.
rdd=spark.sparkContext.parallelize(data)	This will create an RDD from the data frame created by the user. RDD is like a "list" in python
dataframe=rdd.toDF()	This command converts RDD to data frame.
dataframe.show()	This command displays the data frame created or updated.
rdd.collect()	Retrieves the RDD/Data frame and all its elements. It is an action operation of RDD.
count(), first(), max(), reduce(), take(), collect()	Actions of RDD.
flatMap(), map(), reduceByKey(), filter(), sortByKey()	Transformations of RDD.
printSchema()	Prints the schema of the data frame along with the column name and data type.
filter()	Function is used to filter the rows based on the given condition or SQL expression.
select()	Function selects and displays the column. Multiple columns can also be selected by writing column names separated by ","
withColumn()	This function is used to perform changes on

	the existing column (changing datatype, changing value, creating new column).
withColumnRenamed()	This function helps to rename any column.
where()	This can be used instead of filter() by giving SQL expressions or conditions inside the brackets.
>=, ==, <=, !, !=	These are the Boolean conditions that can be used inside filter(), where(), and also in the 'CASE' .
orderBy() and sort()	Both are similar. Used to sort the columns values of data frame in ascending or descending order.
groupBy()	Used to collect identical rows of the group and perform operations like sum, average, mean, maximum and minimum.
join()	Joins the two different data frames or more by column wise.
union(), unionAll(), unionByname()	Merges the two data frames or more having the same structure and schema.
sample(withReplacement, fraction, seed=None)	Creates subset from the large data frame depending upon the fraction mentioned
partitionBy()	Splits the large datasets into smaller depending upon the partition keys.
split()	This function splits the column into two or more if the values inside a column separated by blank or comma.
dropDuplicates()	This function drops the rows with duplicate/same values.
dataframe.na.drop()	Drops/removes the null/missing values from the dataset.
dataframe.na.fill()	Fills the null/missing values with any other value in the dataset.
alias()	Similar to withColumnRenamed(), used to give new name to the columns of the datasets that are more understandable.
Window.partitionBy()	This function is operated on groups of row with similar values and returns a single value for the group of rows.

sum(), count(), max(), min(), mean(), avg(),	Aggregative functions used along with Window functions, groupBy(). Used whenever we need to perform these operations on the group of rows.
expr()	This function is imported from the SQL functions and used to perform SQL expression on the dataset.
when(“condition”).otherwise(“default”)	When() is imported from pyspark.sql.functions and returns the value or column when the conditions are met.

Section 8.0. References

i. https://sparkbyexamples.com/

ii. https://github.com/spark-examples

iii. https://www.databricks.com/glossary/what-is-parquet

iv. https://spark.apache.org/docs/latest/api/python/

v. https://www.geeksforgeeks.org/get-number-of-rows-and-columns-of-pyspark-dataframe/

vi. https://youtube.com/playlist?list=PLZoTAELRMXVNjiiawhzZ0afHcPvC8jpcg

vii. https://colab.research.google.com/drive/1RlcaGDHQnNRvJx5bjDh4_vWTdnfGXCho#scrollTo=gz4giLzCDV3a&uniqifier=1

viii. https://www.interviewbit.com/blog/apache-spark-architecture/

ix. https://www.analyticsvidhya.com/blog/2021/08/data-preprocessing-in-data-mining-a-hands-on-guide/

x. https://medium.com/@sahu.milan1988/use-apache-spark-without-installing-on-your-machine-cc3125269adb

xi. https://www.kaggle.com/code/tientd95/advanced-pyspark-for-exploratory-data-analysis

xii. https://www.utilityproducts.com/line-construction-maintenance/article/16002449/the-holistic-approach-proactive-voltage-monitoring-drives-longterm-payoff