COMPLETE PySpark with ME

- Nishant Kolapkar

1. What is Big Data and How it Started

- Cobol is the first Business Data Processing Language it allowed to store data in files, create index files, and process data efficiently. (1959)
- Then mySQL Oracle(1977) MSSQL comes in picture.
- RDBMS – What do they offer?
    - SQL- An easy Data Query Language
    - Scripting Languages such as PL/SQL and Transact SQL
    - Interface for other programming language such as JDBC and ODBC(interact with data with languages)

- Data Categories
    - Structured (rows and commas)
    - Semi-Structured

XML vs. **JSON**

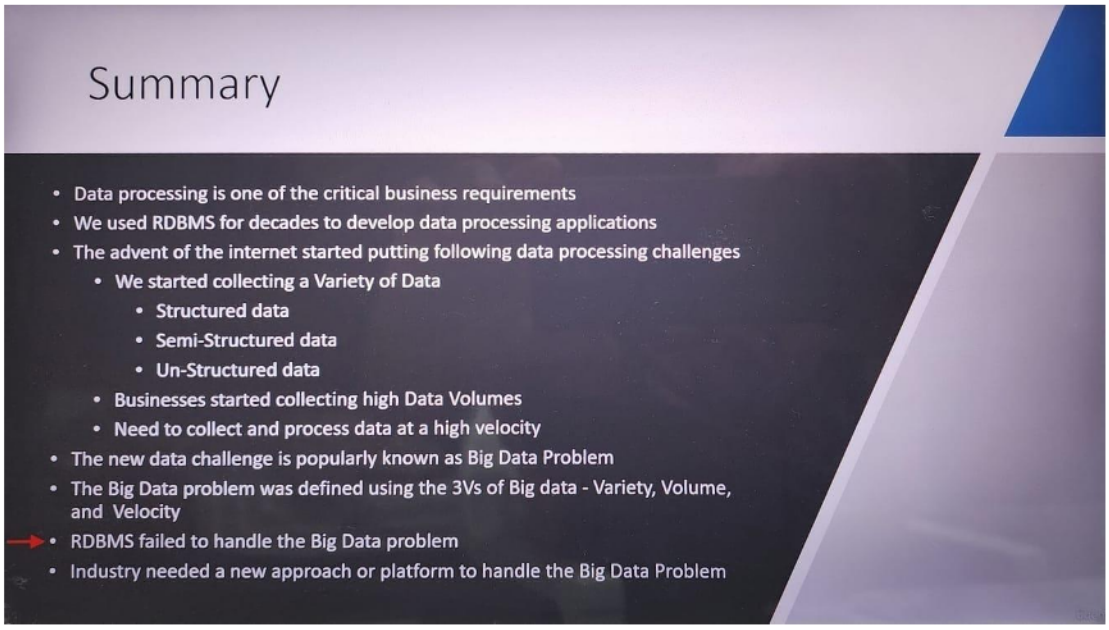
```xml
<?xml version="1.0" encoding="UTF-8"?>
<endereco>
    <cep>31270901</cep>
    <city>Belo Horizonte</city>
    <neighborhood>Pampulha</neighborhood>
    <service>correios</service>
    <state>MG</state>
    <street>Av. Presidente Antônio Carlos, 6627</street>
</endereco>
```

```json
{
    "endereco": {
        "cep": "31270901",
        "city": "Belo Horizonte",
        "neighborhood": "Pampulha",
        "service": "correios",
        "state": "MG",
        "street": "Av. Presidente Antônio Carlos, 6627"
    }
}
```

    - Un-Structured

- Big Data Problem
    - Variety of unstructured Data
    - Huge Amount of Data Volume
    - Velocity (Huge data developed in few sec.)

- Bid Data Platform Requirements
    - Store High Volumes of data arriving at higher velocity.
    - Accommodate structured, semi-structured, and unstructured data variety.
    - Process High volume of a variety of data at a higher velocity.



Approaches of Big Data Solution

Criteria

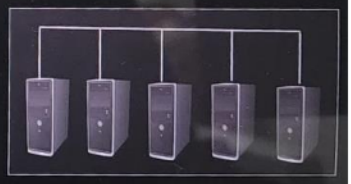
1. Scalability
2. Fault Tolerance and HA
3. Cost-Effectiveness

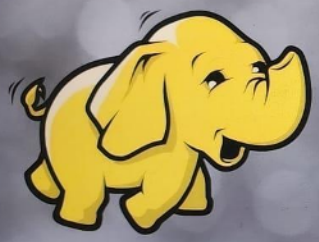
Monolithic Approach

1. Vertical
2. Primary/Secondary
3. Expensive

Distributed Approach

1. Horizontal
2. Multifold
3. Economical





Hadoop Platform

YARN - Cluster Operating System

HDFS - Distributed Storage

Map/Reduce – Distributed Computing

Databases and Hadoop

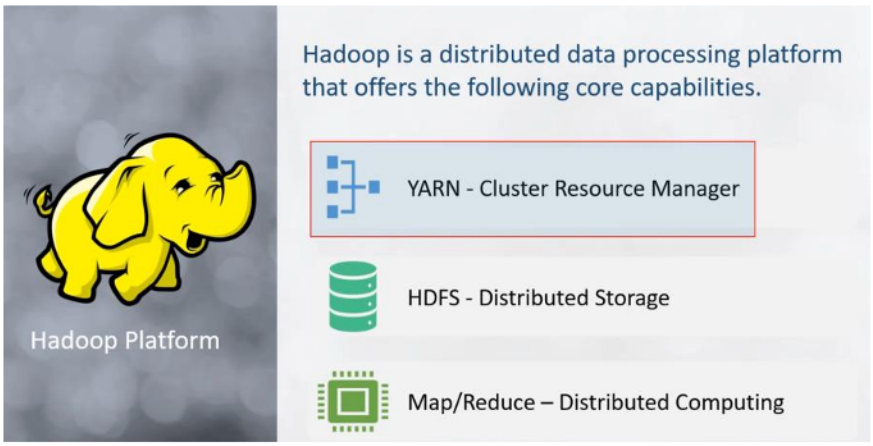
RDBMS Database

1. Data storage
2. SQL Query language
3. Scripting language
4. Programming languages Interface

Hadoop

1. Data storage
2. Hive SQL Query language
3. Pig Scripting language
4. Programming languages Interface

- What is Hadoop?

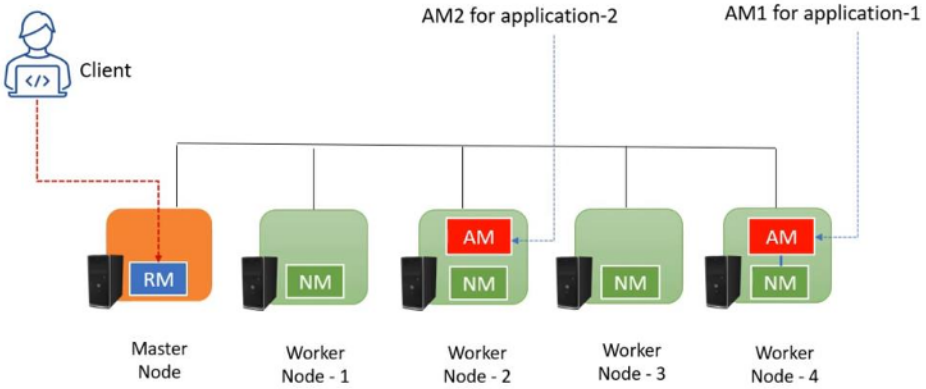


- YARN is Hadoop Cluster OS but it called as Cluster Resource Manager.

-

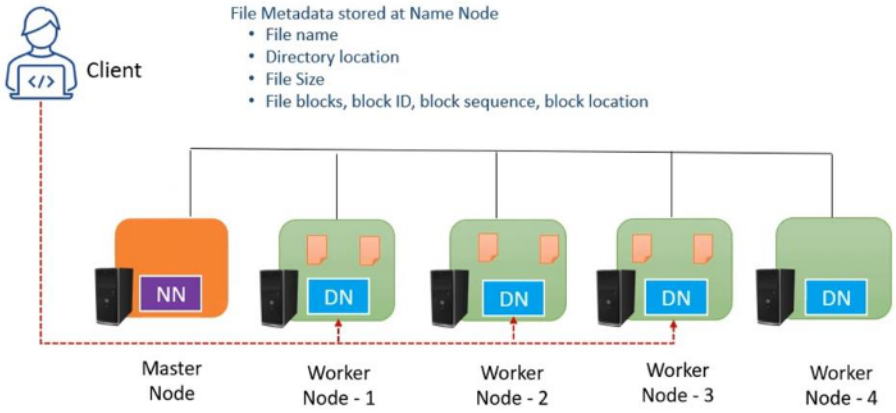
YARN – Yet Another Resource Manager

- Hadoop cluster Operating System
- Popularly known as Hadoop Cluster Resource Manager
- Has three main components
  - RM - Resource Manager
  - NM - Node Manager
  - AM - Application Master



HDFS – Hadoop Distributed File System

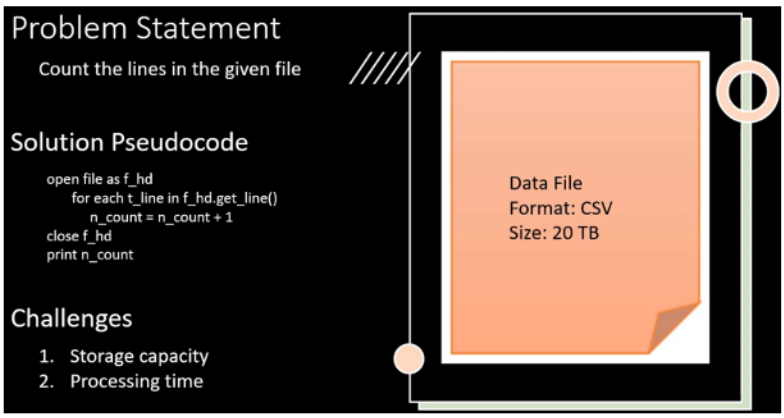
- Distributed Storage on Hadoop Cluster
- Has two main components
  - NN – Name Node
  - DN – Data Node

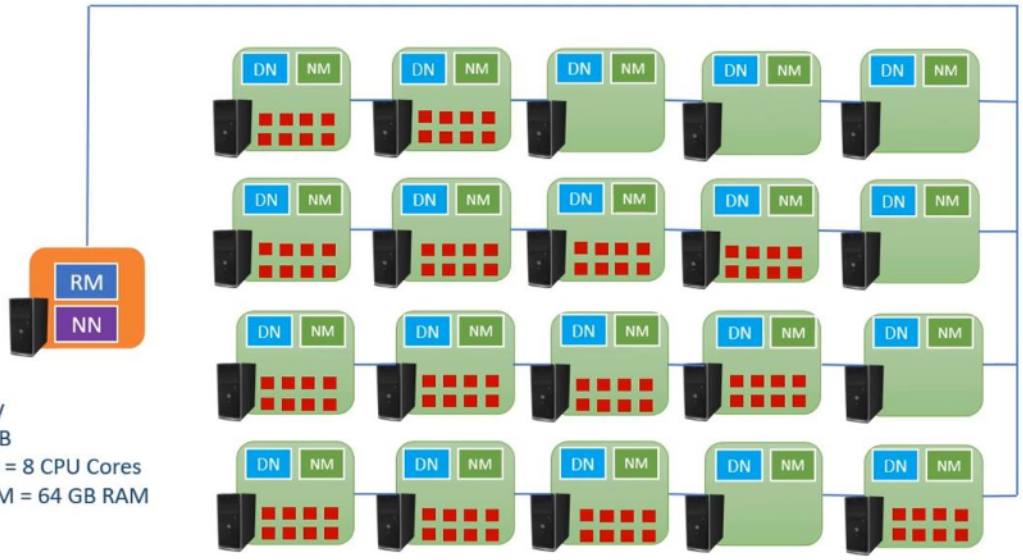


Data will be split into blocks and stored on nodes normaly this blocks size is 128mb. And this all information of blocks are stored in Metadata and stored at Name Node. For read operation by using this metadata info all blocks are combined and retrive to client from cluster.

M/R – Hadoop Map Reduce Framework

- Map Reduce is
  - Programing Model
  - Programing Framework





Node Capacity
4 x 2 TB = 8 TB
4 x Dual Core = 8 CPU Cores
4 x 16 GB RAM = 64 GB RAM

Problem Statement

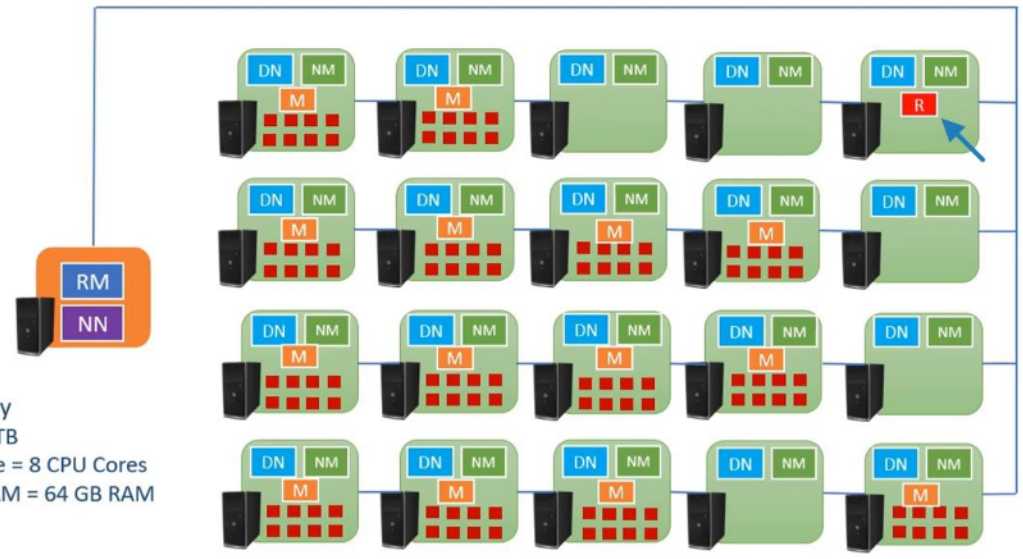
Count the lines in the given file

Distributed Solution Pseudocode

```
def map(file_block):
    open file_block as fb_hd
    for each t_line in fb_hd.get_line()
        n_count = n_count + 1

    close fb_hd
    return n_count
```

```
def reduce(list_counts):
    for each cnt in list_counts
        total_count = total_count + cnt

print total_count
```

Data File
Format: CSV
Size: 20 TB



Node Capacity
4 x 2 TB = 8 TB
4 x Dual Core = 8 CPU Cores
4 x 16 GB RAM = 64 GB RAM

Summary

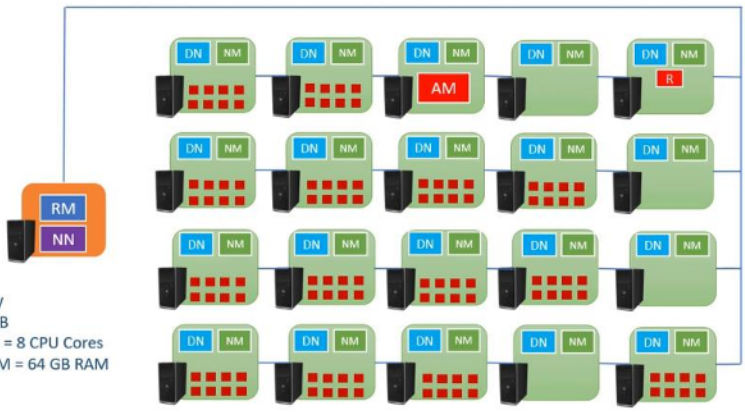
Map Reduce Model

Implement logic in two functions

1. Map Function
   - Reads data block
   - Applies logic at block level
   - Map output is sent to Reduce
2. Reduce Function
   - Receives Map output
   - Consolidates the results

Hadoop M/R framework implement the map-reduce model.

- YARN manages resource allocation
- HDFS manages data blocks



Node Capacity
4 x 2 TB = 8 TB
4 x Dual Core = 8 CPU Cores
4 x 16 GB RAM = 64 GB RAM

>>

Hive capabilities

1. Create
   1. Databases
   2. Tables
   3. Views
2. Run SQL Queries

Entering Apache Spark

Advantages over Hadoop

1. Performance
   - 10 to 100 times faster than Hadoop M/R
2. Ease of development
   - Spark SQL
   - High performance SQL Engine
   - Composable Function API
3. Language support
   - Java, Scala, Python and R
4. Storage
   - HDFS Storage
   - Cloud Storage
5. Resource Management
   - YARN, Mesos, Kubernetes

APACHE Spark™

Runs in two setups
1. With Hadoop (Data Lake)
2. Without Hadoop (Lakehouse)

Spark and Databricks Cloud

Spark Solutions

1. Data Lake – On Hadoop
2. Lakehouse – On Cloud

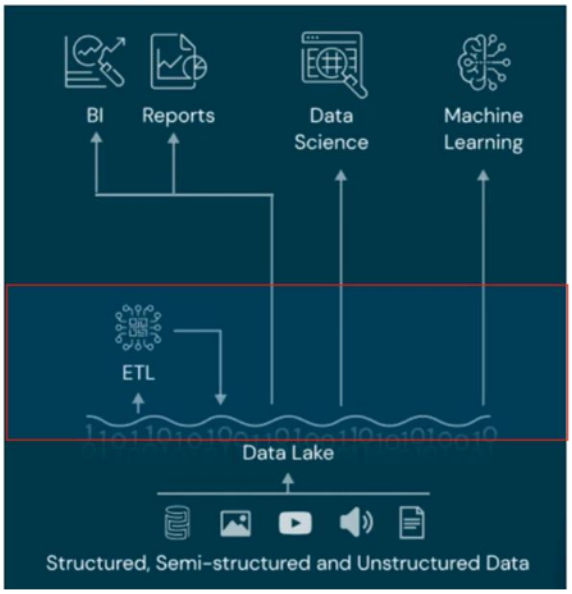
APACHE Spark™ + Databricks

- What is Data Lake and How it works
  - Google File System(GFS) >> Hadoop Distributed File System(HDFS with Map Reduce)

1. Vertical Scaling
2. Large Capital Investment
3. Terabytes to Exabytes
4. Only Structured Data

1. Horizontal Scaling
2. Low Capital Investment
3. Exabytes to Petabytes
4. Structured, Semi Structured and Un Structured Data

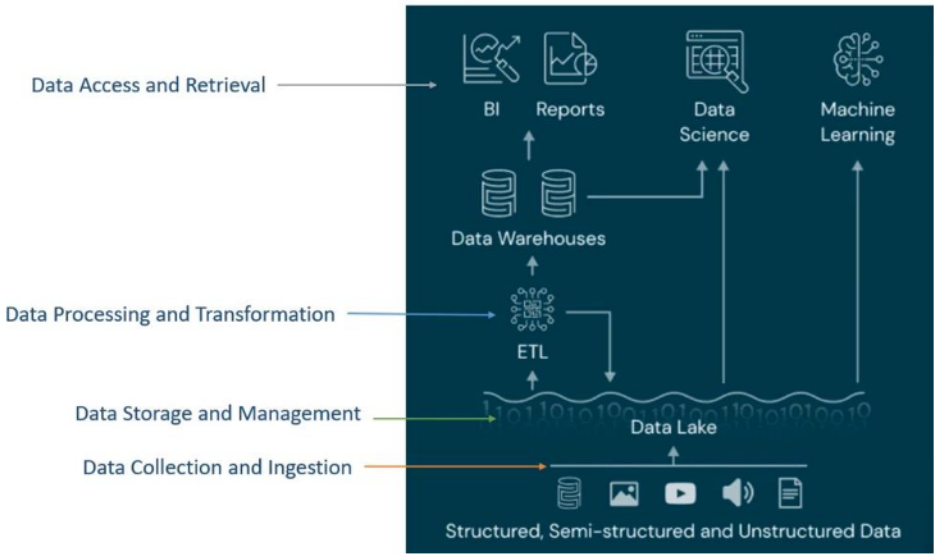
Data Lake



Key Missing Features

1. Transaction and Consistency
2. Reporting Performance

Data Lake



Data Storage and Management

HDFS Amazon S3 Azure Blob Google Cloud

Data Processing and Transformation

1. Initial data quality check.
2. Transforming and preparing data.
3. Correlating, Aggregating, Analysing, and extracting business insights.
4. Applying machine learning model.

Apache Spark

Data Access and Retrieval

Data Scientist
Rest Interface
File Download
JDBC/ODBC
Search

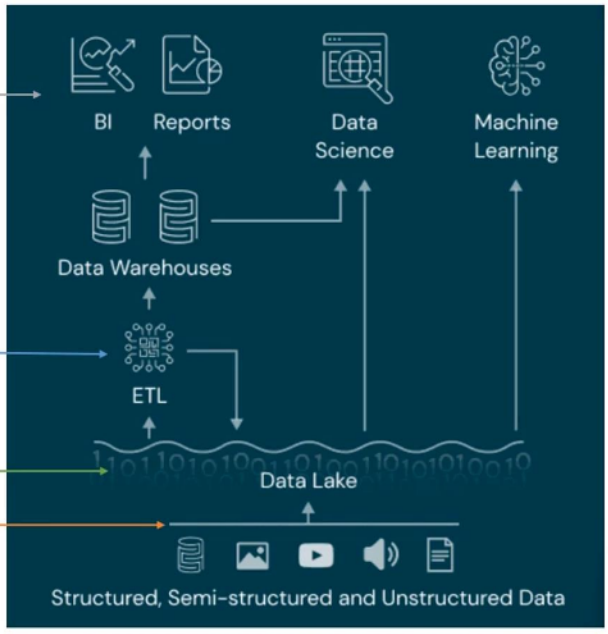
Data Lake

Data Access and Retrieval

Data Processing and Transformation

Data Storage and Management

Data Collection and Ingestion



Data Scientist
Rest Interface
File Download
JDBC/ODBC
Search

Apache Spark

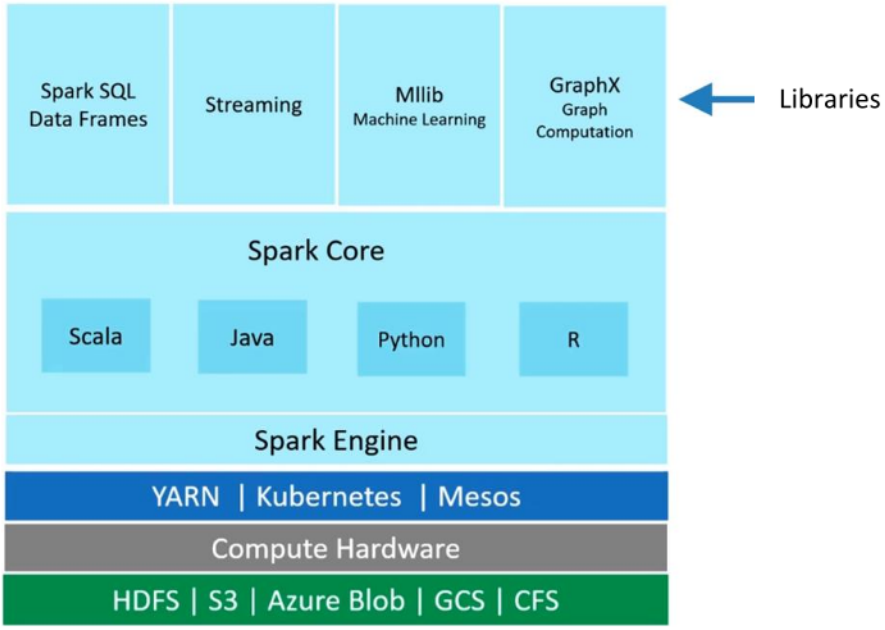
HDFS Amazon S3 Azure Blob Google Cloud

HVR Azure Data Factory Informatica talend

Security | Workflow | Metadata | Data Lifecycle | Monitoring | Operations

- Introducing Apache Spark and Databricks Cloud

  - [What is Apache Spark?]



1. Spark on the Cloud Platform
2. Spark Cluster Management
3. Notebooks and Workspace
4. Administration Controls
5. Optimized Spark
6. Databases/Tables and Catalog
7. Databricks SQL Analytics
8. Delta Lake Integration
9. ML Flow
10. Industry vertical accelerators

- Installation Using Apache Spark

1. Cloud Platforms
   - Notebook
   - Databricks Cloud
2. On-premise Platforms
   - Python IDE
   - Cloudera Platform

ref. 8

```
Microsoft Windows [Version 10.0.19045.2486]
(c) Microsoft Corporation. All rights reserved.

C:\Users\innk>setx JAVA_HOME "C:\Program Files\Java\jdk-11"

SUCCESS: Specified value was saved.

C:\Users\innk>setx PATH "%PATH%;%JAVA_HOME%\bin"

WARNING: The data being saved is truncated to 1024 characters.

SUCCESS: Specified value was saved.

C:\Users\innk>_
```

```
C:\Users\innk>java -version
openjdk version "11" 2018-09-25
OpenJDK Runtime Environment 18.9 (build 11+28)
OpenJDK 64-Bit Server VM 18.9 (build 11+28, mixed mode)
```

```
C:\Users\innk>setx HADOOP_HOME "E:\pySpark_soft\hadoop-3.2.2"

SUCCESS: Specified value was saved.
```

SPARK_HOME E:\pySpark_soft\spark_3

E:\pySpark_soft\hadoop-3.2.2\bin
E:\pySpark_soft\spark_3\bin

```
C:\Users\innk>setx PYTHONPATH "E:\pySpark_soft\spark_3\python;E:\pySpark_soft\spark_3\python\libpy4j-0.10.9.5-src.zip"

SUCCESS: Specified value was saved.

C:\Users\innk>where python
C:\Program Files\Python311\python.exe

C:\Users\innk>setx PYSPARK_PYTHON "C:\Program Files\Python311\python.exe"

SUCCESS: Specified value was saved.
```

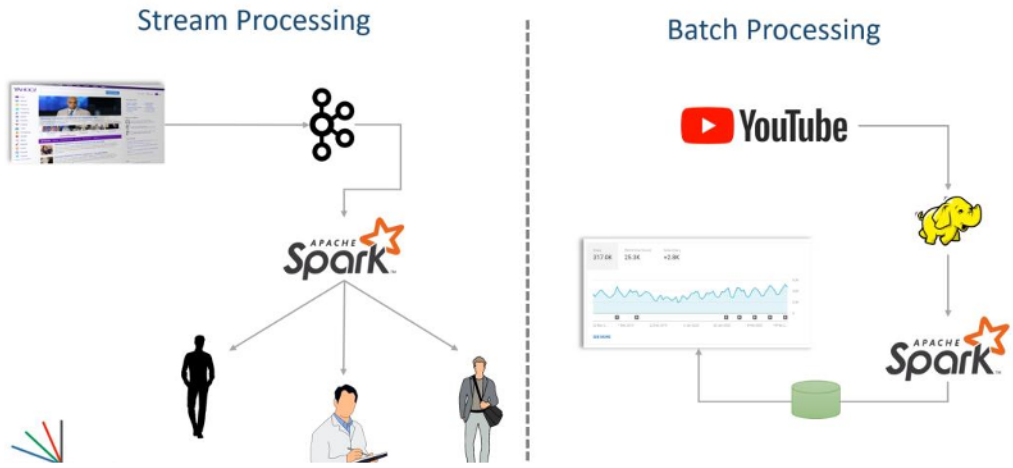
Need to install python v3.10.9

- **Spark Execution Model and Architecture**

  - What is Apache Spark? A Distributed Computing Platform.
  - What do we do with Spark? We create program and execute them on Cluster.
    - How to create Spark Programs?
    - How to execute Spark Programs?

How to execute Spark Programs?

1. Interactive Clients
   spark-shell, Notebook



For both scenarios output appears on dashboard.

2. Submit Job
   spark-submit (allow to submit job to the cluster)

How to execute Spark Programs?

1. Interactive Clients
   spark-shell, Notebook
2. Submit Job
   spark-submit, Databricks Notebook, Rest API

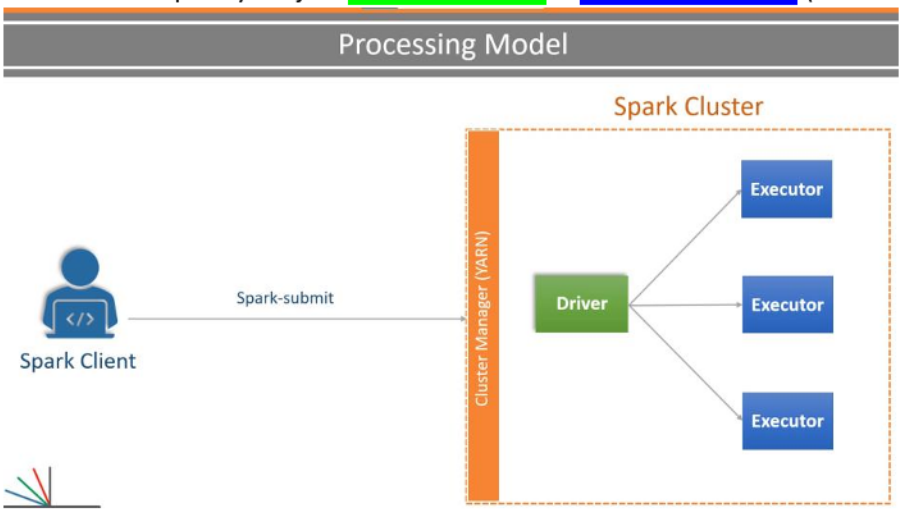
- Interactive clients are used for learning, Dev and Explorations.

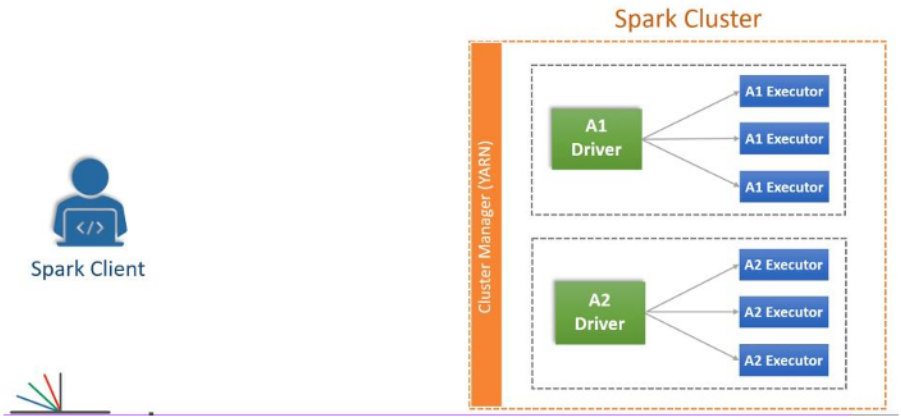
- Packaging of the Spark applications and submitting on the cluster for execution.

- Spark Distributed Processing Model – How your program runs?

Spark Distributed Processing Model?

- Spark applies to a master slave architecture to every application. When we submit application its create master process for the application and this master process is create bunch of slaves to distribute the work and compute your job. Master = Drivers & Slaves = Executors (with respect to clusters its different)



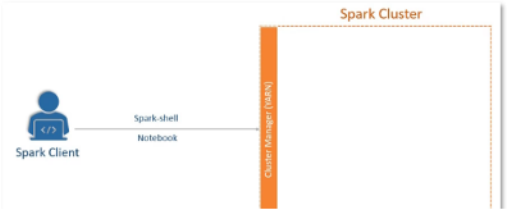


- Every Spark applications applies a master slave architecture and runs independently on the cluster.

- Spark Execution Modes and Cluster Managers (What happened after you submit Spark Application?)

How Spark works on Local Machine?

How Spark works with Interactive Clients?



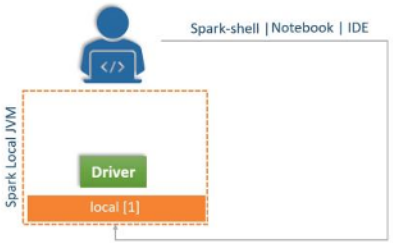
- We can run Spark application on local without having cluster
- Configuration application to run on verity of Clusters Spark is compatible with following Clusters.

1. local[n]
2. YARN
3. Kubernetes
4. Mesos
5. Standalone

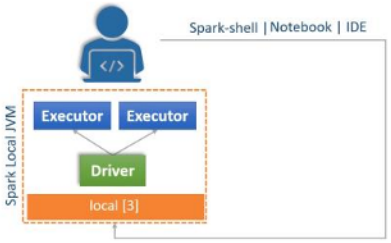
>>

```
spark.conf
1  [SPARK_APP_CONFIGS]
2  spark.app.name = HelloSpark
3  spark.master = local[3]
4  spark.sql.shuffle.partitions = 2
```

Here is configuration which tells about the target cluster manager



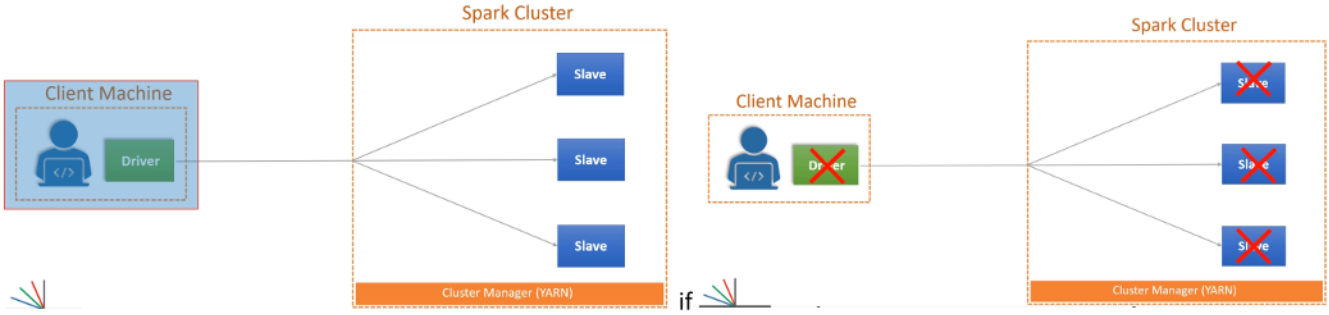
OR



(running testing debugging application locally)

How to Spark with interactive clients?

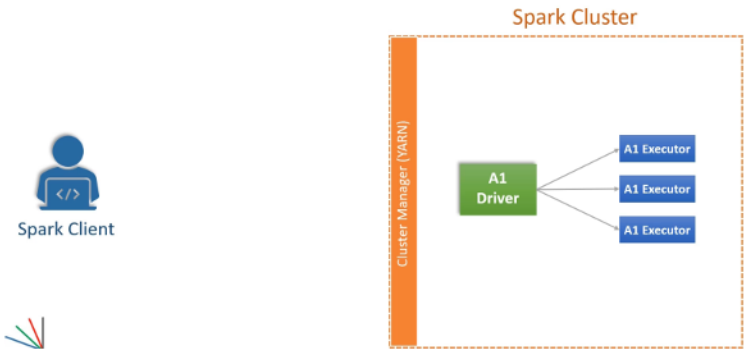
1. Client mode



if

Slaves also dies of the absence of the driver. Its suitable for interactive work and not for long running jobs.

2. Cluster mode



In this mode every thing is runs on cluster when you log of client machine your driver is not impacted and slaves because its running on the cluster.

- Three widely used execution models

Cluster Managers	Execution Modes	Execution Tools
1. local[n]	1. Client	1. IDE, Notebook
2. YARN	2. Client	2. Notebook, Shell
3. YARN	3. Cluster	3. Spark Submit

- **Working with PySpark Shell**

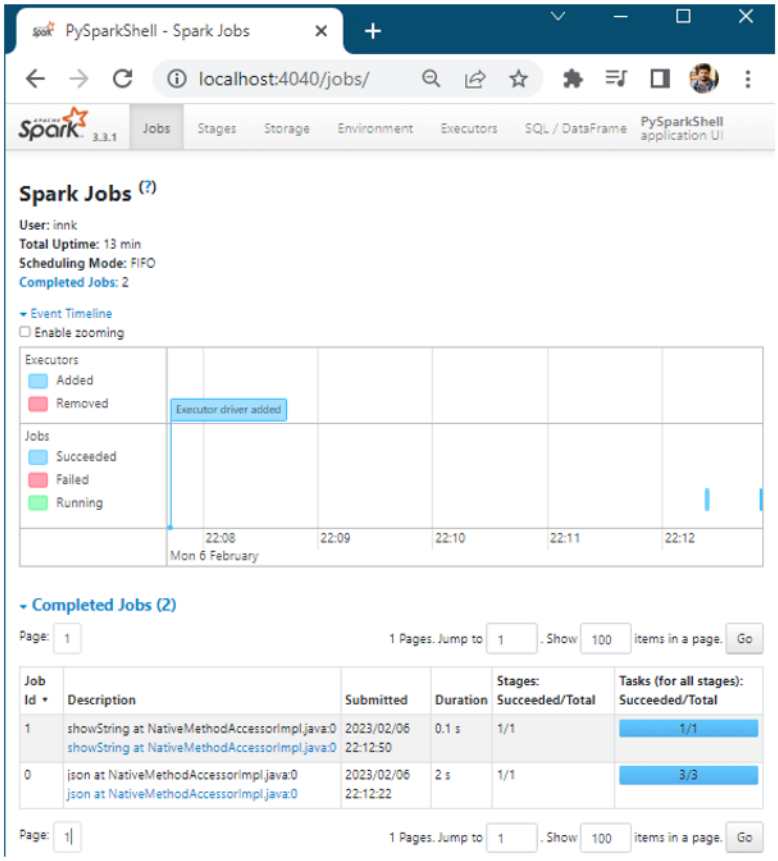
Cluster	Mode	Tool
Local	Client Mode	spark-shell,

- Spark Shell on local Client Mode:
  pyspark –help

```
>pyspark --master local[3] --driver-memory 2G
```
To increase memory to 2GB (only for 1st time)

```
>>> df = spark.read.json("C:/demo/notebook/data/people.json")
>>> df.show()
+----+--------+
| age|    name|
+----+--------+
|null|Prashant|
|  30|   Abdul|
|  19|  Justin|
|  43|    Andy|
+----+--------+
```

- Spark context Web UI:
  http://localhost:4040/jobs/ (This UI is available only when Spark application is in running state)



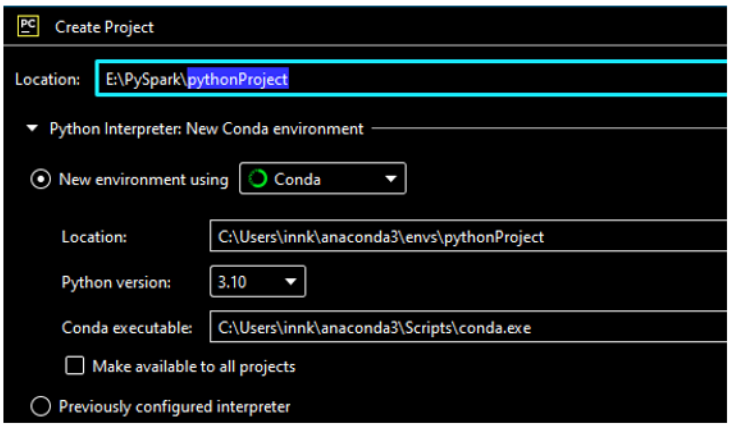
- Installing Multi-Node Spark Cluster
  How to setup Spark Cluster in GCP?

- **YARN Client Mode Using Spark-Shell** (Directly on production cluster, here GCP) ref. 21, 22

Cluster	Mode	Tool
YARN	Client Mode	spark-shell, Notebook

YARN Dynamic policy | Zeppelin

- **PySpark Project One**



use Conda environment with Python 3.10

pip install pyspark

pip install pytest

- **Spark Application Logs**

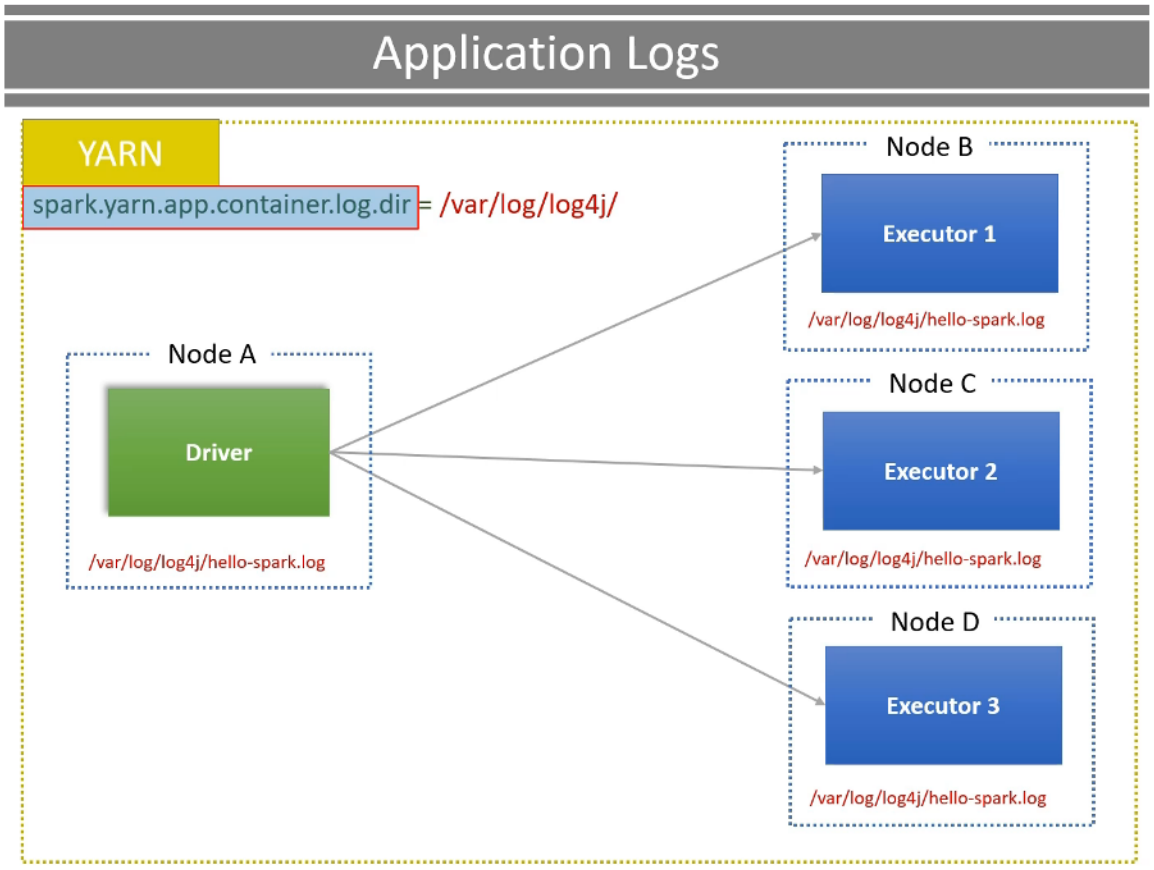
Using Log4J
1. Create a Log4J configuration file
2. Configure Spark JVM to pickup the Log4J configuration file
3. Create a Python Class to get Spark's Log4J instance and use it

Instead of python built in logger why Log4J is used?

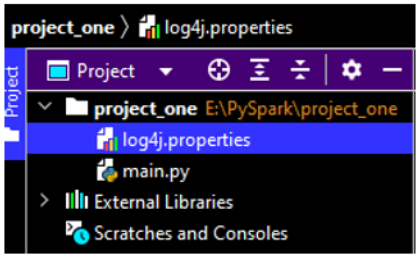
Because the Python Logger is not integrated with Spark.

Using Log4J
→ 1. Logger
2. Configurations
3. Appender

ref. 25



YARN only collects logs only from /var/log/log4l/location



copy log4j.properties to main project folder

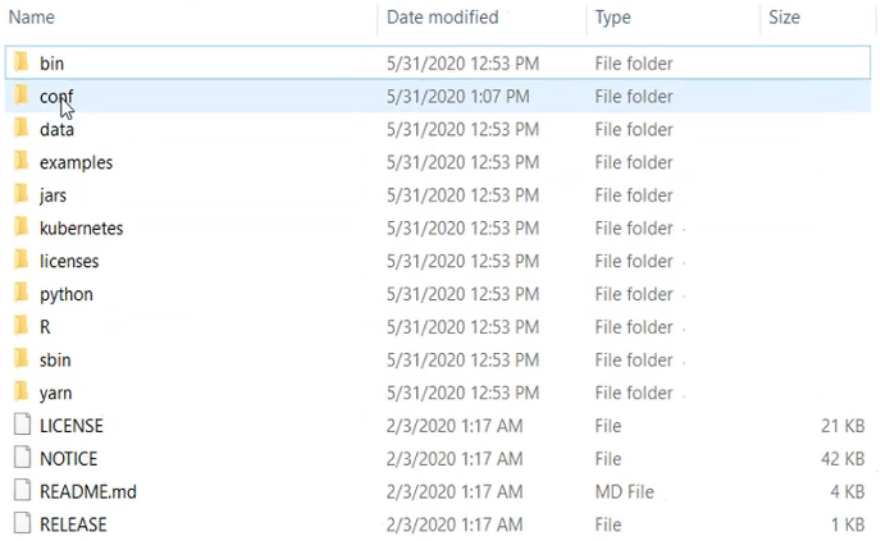
```
#define rolling file appender
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=${spark.yarn.app.container.log.dir}/${logfile.name}.log
```

How to configure JVM variables?

Java Virtual Machine

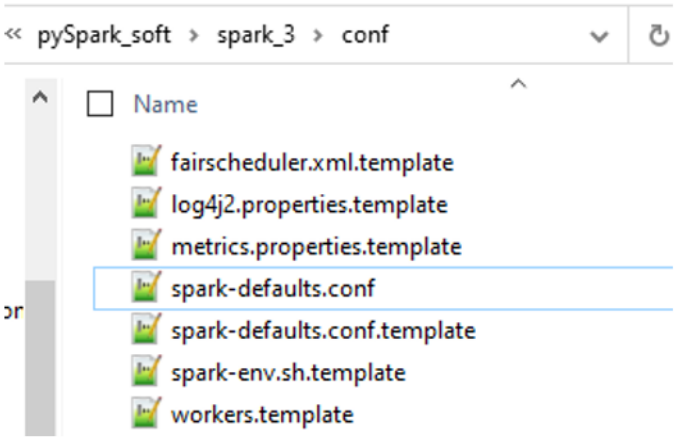
Setting JVM Parameters

SPARK_HOME/conf/spark-defaults.conf



Name	Date modified	Type	Size
bin	5/31/2020 12:53 PM	File folder	
conf	5/31/2020 1:07 PM	File folder	
data	5/31/2020 12:53 PM	File folder	
examples	5/31/2020 12:53 PM	File folder	
jars	5/31/2020 12:53 PM	File folder	
kubernetes	5/31/2020 12:53 PM	File folder	
licenses	5/31/2020 12:53 PM	File folder	
python	5/31/2020 12:53 PM	File folder	
R	5/31/2020 12:53 PM	File folder	
sbin	5/31/2020 12:53 PM	File folder	
yarn	5/31/2020 12:53 PM	File folder	
LICENSE	2/3/2020 1:17 AM	File	21 KB
NOTICE	2/3/2020 1:17 AM	File	42 KB
README.md	2/3/2020 1:17 AM	MD File	4 KB
RELEASE	2/3/2020 1:17 AM	File	1 KB

This is the place where Spark is looking for the configuration dir. SPARK_HOME



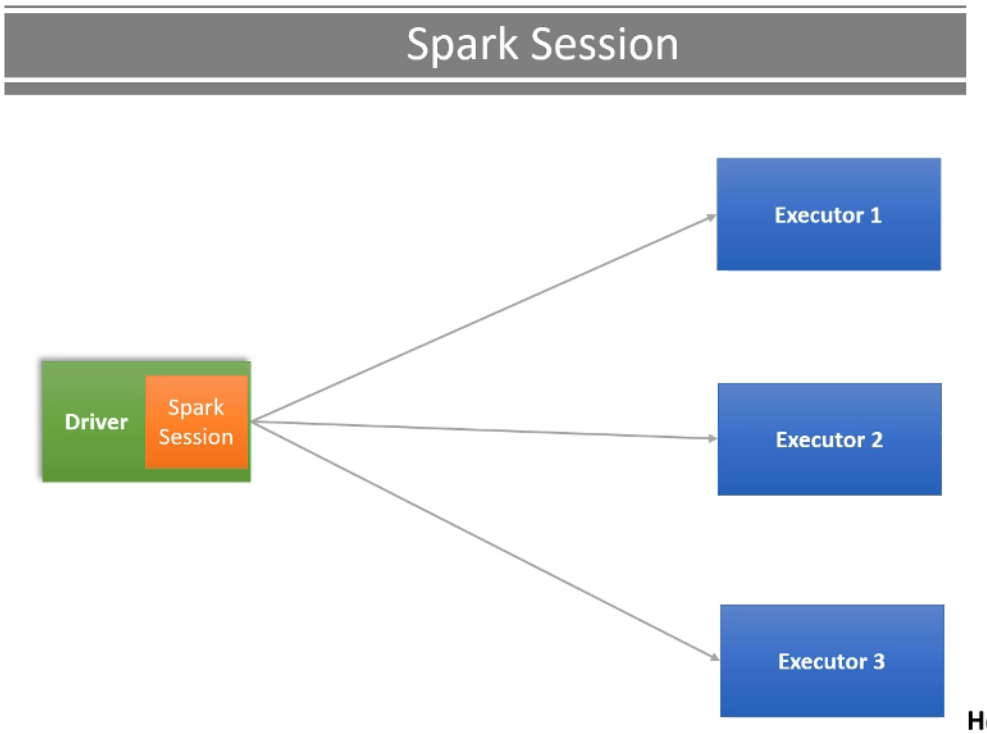
spark-defaults.conf

spark.driver.extraJavaOptions -Dlog4j.configuration=file:log4j.properties -DSpark.yarn.app.container.log.dir=app-logs -Dlogfile.name=First_Spark

spark-defaults.conf

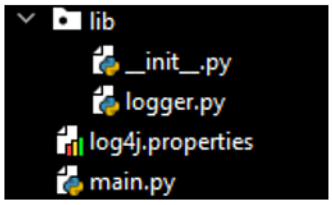
```
#
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or impli
# See the License for the specific language governing permissions and
# limitations under the License.
#

# Default system properties included when running spark-submit.
# This is useful for setting default environmental settings.

# Example:
# spark.master                     spark://master:7077
# spark.eventLog.enabled           true
# spark.eventLog.dir               hdfs://namenode:8021/directory
# spark.serializer                 org.apache.spark.serializer.KryoSeri
# spark.driver.memory              5g
# spark.executor.extraJavaOptions  -XX:+PrintGCDetails -Dkey=value -Dnu
spark.driver.extraJavaOptions      -Dlog4j.configuration=file:log4j.pro
```

- **Creating Spark Session**



Here Spark Session is Driver

Steps :



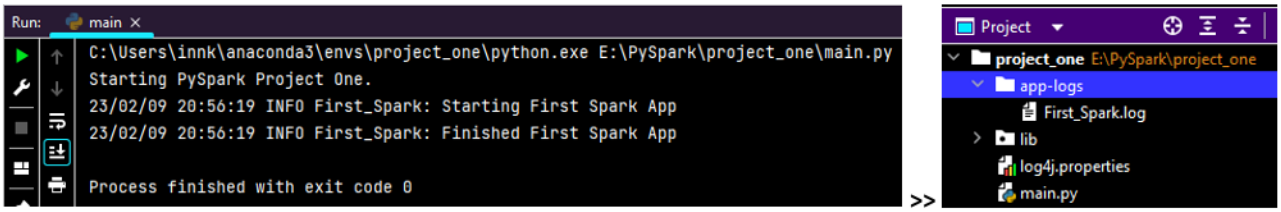
create package name lib at project level create logger.py

```
class Log4j(object):
    def __init__(self, spark):
        root_class = "guru.learningjournal.spark.examples"
        conf = spark.sparkContext.getConf()
        app_name = conf.get("spark.app.name")
        log4j = spark._jvm.org.apache.log4j
        self.logger = log4j.LogManager.getLogger(root_class + "." + app_name)

    def warn(self, message):
        self.logger.warn(message)

    def info(self, message):
        self.logger.info(message)

    def error(self, message):
        self.logger.error(message)

    def debug(self, message):
        self.logger.debug(message)
```

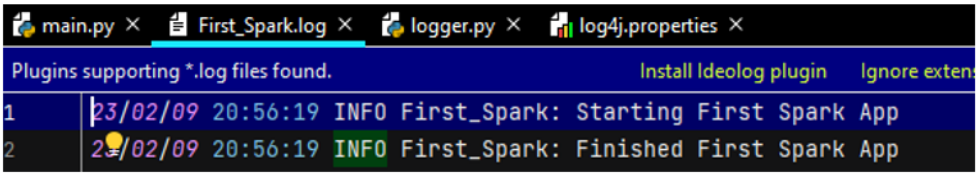
logger.py

main.py

```python
from pyspark.sql import *
from lib.logger import Log4j

if __name__ == "__main__":
    print("Starting PySpark Project One.")

    spark = SparkSession.builder \
        .appName("First_Spark") \
        .master("local[3]") \
        .getOrCreate() \

    logger = Log4j(spark)
    logger.info("Starting First Spark App")
    # Your processing code

    logger.info("Finished First Spark App")
    spark.stop()
```



New app log folder is created after executing main.py



First_Spark.log

Question 1:

SparkSession

- ○ is an entry point to Programming Spark
- ○ can be created using SparkSession.builder()
- ◉ both 1 and 2
- ○ none of the above

Question 2:

A SparkContext

- ○ represents the connection to a Spark cluster
- ○ can be used to get SparkConf object
- ○ was an entry point to Programming Spark in older versions
- ◉ All of the above

Question 3:

SparkConf

- ○ is used to get/set various Spark parameters as key-value pairs
- ○ can be retrieved from your spark session using spark.sparkContext.getConf()
- ○ You can create a SparkConf() that loads defaults from system properties and the classpath
- ○ setting values to SparkConf directly takes priority over system properties
- ◉ All of the above

Question 4:

You can set the configuration to your Spark application

- ○ using spark-submit command-line options
- ○ setting environment variable on the machine where you run spark-submit
- ○ using SPARK_HOME/conf/spark-defaults.conf on the machine where you run spark-submit
- ○ setting key/value pair to SparkConf object in your application code
- ◉ all of the above

Question 5:

Following Spark application configuration precedence is correct

- ○ environment variable -> spark-submit command line -> SparkConf -> spark-defaults.conf
- ○ spark-submit command line -> environment variable -> spark-defaults.conf -> SparkConf
- ○ environment variable -> spark-submit command line -> spark-defaults.conf -> SparkConf
- ◉ environment variable -> spark-defaults.conf -> spark-submit command line -> SparkConf

- **Configuring Spark Session**

Configuring Spark Session

1. Environment variables
2. SPARK_HOME\conf\spark-defaults.conf
3. spark-submit command-line options
4. SparkConf Object

2nd method is used for special cases. 3rd and 4th is used by Dev

3rd

```
Microsoft Windows [Version 10.0.18362.836]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\demo\spark245>bin\spark-submit --master local[3] --conf "spark.app.name=Hello Spark" --conf spark.eventLog.enabled=false HelloSpark.py
```

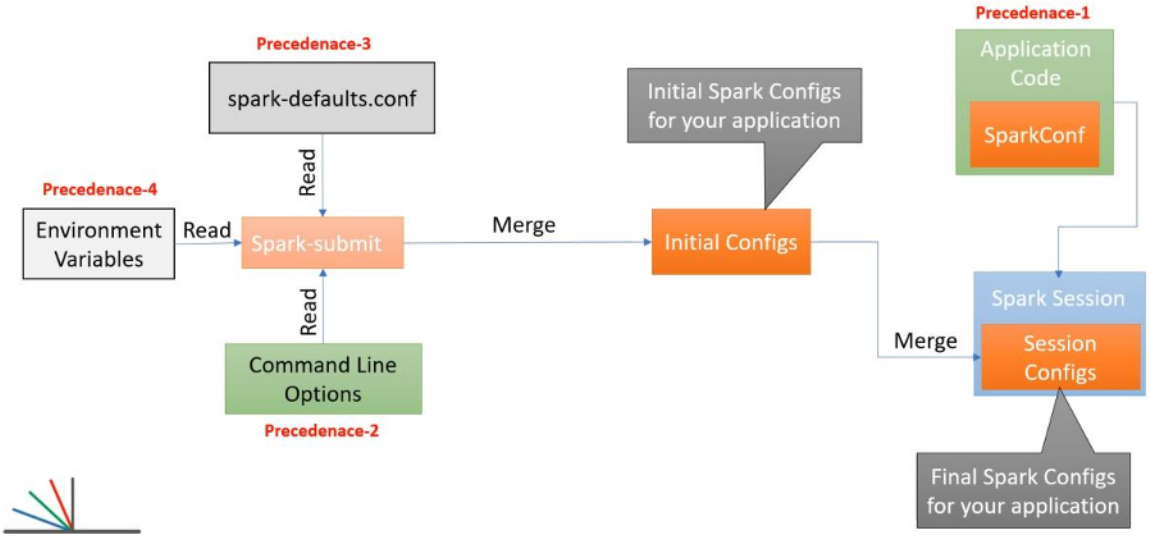
- Config can also possible with above submit command-line which is 3rd one.
- IF you have space in config use “ “.

4th Quote the configuration in Application

https://spark.apache.org/docs/latest/configuration.html#application-properties

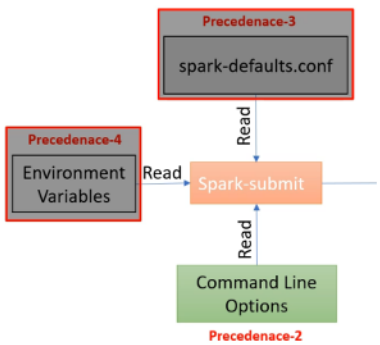
```
from pyspark import SparkConf
from pyspark.sql import *

from lib.logger import Log4J

if __name__ == "__main__":
    conf = SparkConf()
    conf.set("spark.app.name", "Hello Spark")
    conf.set("spark.master", "local[3]")
    spark = SparkSession.builder \
        .config(conf=conf) \
        .getOrCreate()

    logger = Log4J(spark)

    logger.info("Starting HelloSpark")
    # Your processing code

    logger.info("Finished HelloSpark")
    # spark.stop()
```

- **Precedence Levels of config methods**

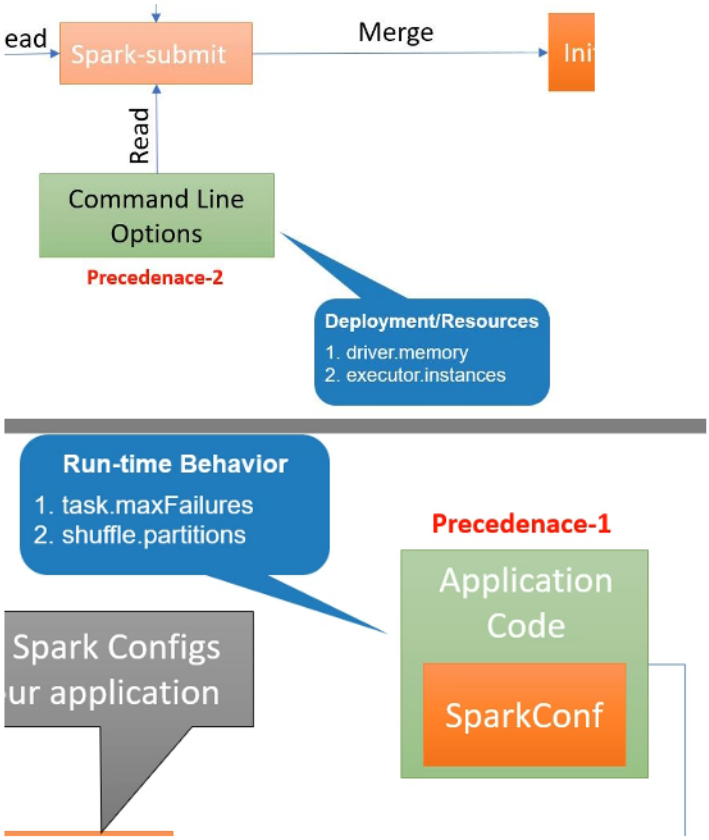


- **Rule :**

  Leave for Cluster Admins



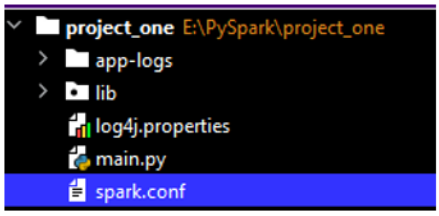
You should be use 3rd command line or 4th Application config



- **Steps for Application level config**

```
from pyspark.sql import *
from pyspark import SparkConf
from lib.logger import Log4j

if __name__ == "__main__":
    conf = SparkConf()
    conf.set("spark.app.name","First Spark")
    conf.set("spark.master","local[3]")
    spark = SparkSession.builder \
        .config(conf=conf)\
        .getOrCreate() \

    logger = Log4j(spark)
    logger.info("Starting First Spark App")
    # Your processing code

    logger.info("Finished First Spark App")
    spark.stop()
```

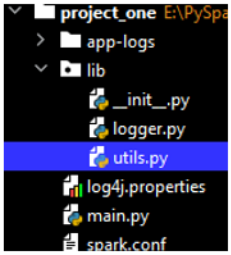
Actually in real world project to minimize complexity and errors go through following



Create spark.conf file

```
[SPARK_APP_CONFIGS]
spark.app.name = Hello Spark
spark.master = local[3]
```

spark.conf



Create util.py in lib package

```python
import configparser
from pyspark import SparkConf

def get_spark_app_config():
#This fun will load the config from spark.conf file and return spark_conf_object
    spark_conf = SparkConf()
    config = configparser.ConfigParser()
    config.read("spark.conf")

    for (key, val) in config.items("SPARK_APP_CONFIGS"):
        spark_conf.set(key,val)
    return spark_conf
```

util.py

```python
from pyspark.sql import *
from lib.utils import get_spark_app_config
from lib.logger import Log4j

if __name__ == "__main__":
    conf = get_spark_app_config()
    spark = SparkSession.builder \
        .config(conf=conf)\
        .getOrCreate() \

    logger = Log4j(spark)
    logger.info("Starting First Spark App")
    # Your processing code

    logger.info("Finished First Spark App")
    spark.stop()
```

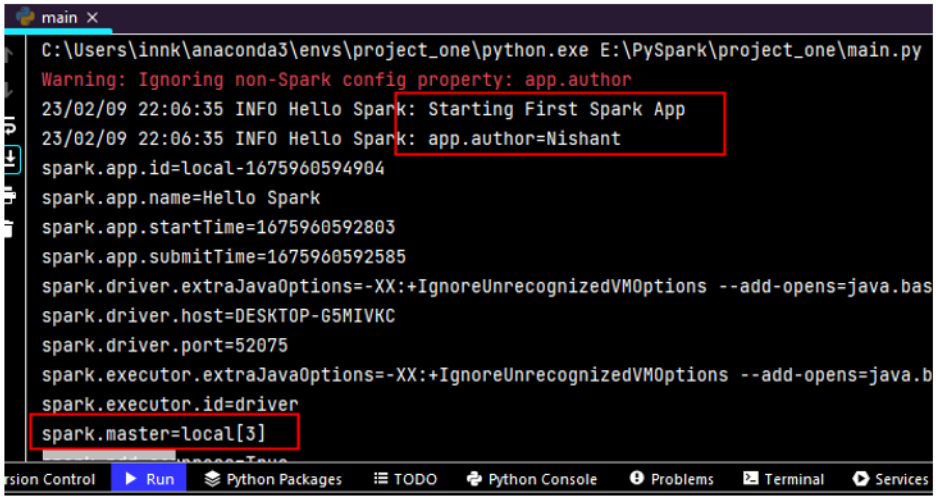
main.py **(here hard coding is fixed)**

```
[SPARK_APP_CONFIGS]
spark.app.name = Hello Spark
spark.master = local[3]
app.author = Nishant
```

spark.conf

```python
from pyspark.sql import *
from lib.utils import get_spark_app_config
from lib.logger import Log4j

if __name__ == "__main__":
    conf = get_spark_app_config()
    spark = SparkSession.builder \
        .config(conf=conf)\
        .getOrCreate() \

    logger = Log4j(spark)
    logger.info("Starting First Spark App")
    # Your processing code

    conf_out = spark.sparkContext.getConf()
    logger.info(conf_out.toDebugString())
    #To see out come

    logger.info("Finished First Spark App")
    spark.stop()
```

main.py

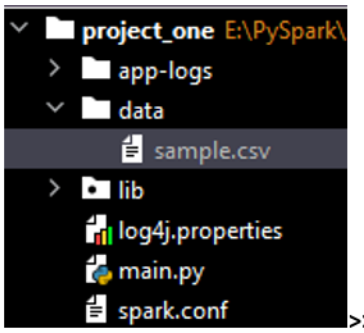


O/P

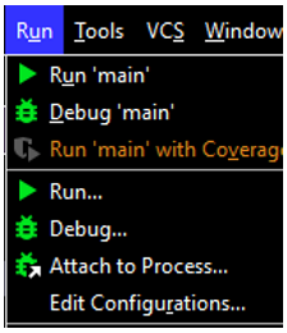
- **Data Frame Introduction**

1. Read
2. Processes
3. Write

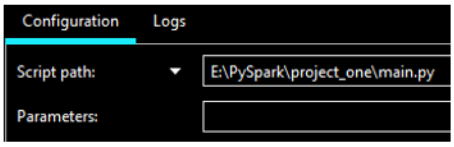
- **Run / Debug Configuration**

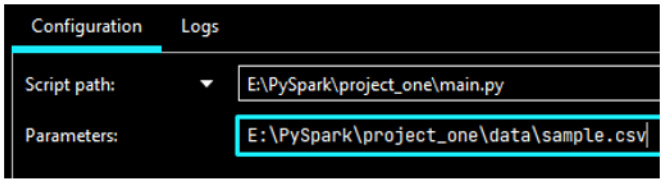


>>



Edit Configuration >>



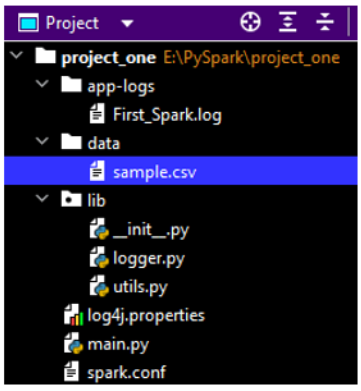


To avoiding hard code n path to file

Timestamp	Age	Gender	Country	state
2014-08-27 11:29:31	37	Female	United States	IL
2014-08-27 11:29:37	44	M	United States	IN
2014-08-27 11:29:44	32	Male	Canada	
2014-08-27 11:29:46	31	Male	United Kingdom	
2014-08-27 11:30:22	31	Male	United States	TX

Data Frame Schema

1. Column Names
2. Data Types



complete project structure for ref.

utils.py

```
import configparser
from pyspark import SparkConf

def get_spark_app_config():
#This fun will load the config from spark.conf file and return spark_conf_object
    spark_conf = SparkConf()
    config = configparser.ConfigParser()
    config.read("spark.conf")

    for (key, val) in config.items("SPARK_APP_CONFIGS"):
        spark_conf.set(key,val)
    return spark_conf

def load_survey_df(spark, data_file):
    return spark.read \
        .option("header", "true") \
        .option("inferSchema", "true") \
        .csv(data_file)\
    # pandas dataframe reader : header, Schema : infraSchema
```

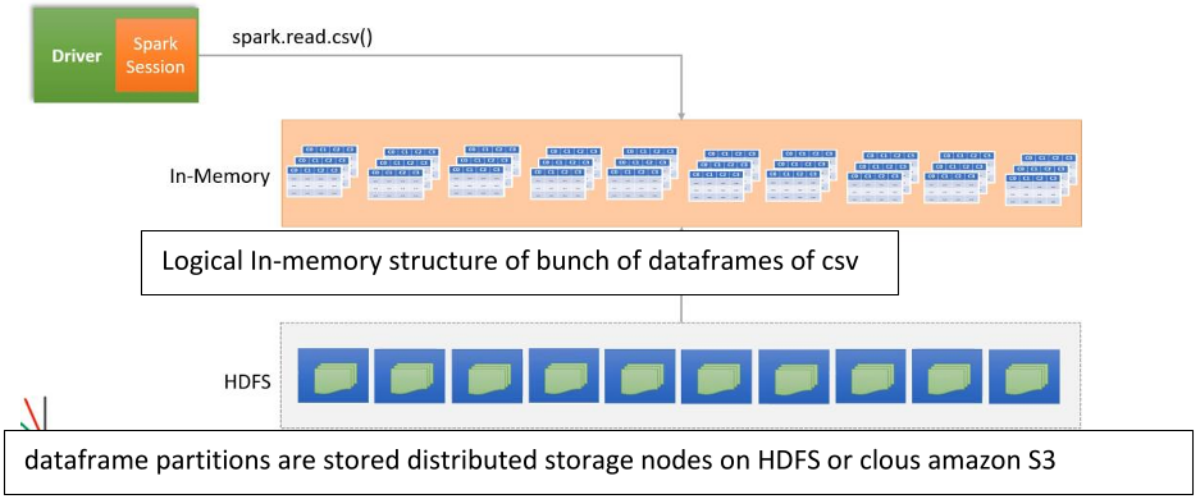
main.py

```
import sys
from pyspark.sql import *
from lib.utils import get_spark_app_config, load_survey_df
from lib.logger import Log4j

if __name__ == "__main__":
    conf = get_spark_app_config()
    spark = SparkSession.builder \
        .config(conf=conf)\
        .getOrCreate() \

    logger = Log4j(spark)

    if len(sys.argv) !=2: # Command line argument if not provided we get error
        logger.error("Usage: HelloSpark <filename>")
        sys.exit(-1)

    logger.info("Starting First Spark App")
    # Your processing code

    survey_df = load_survey_df(spark, sys.argv[1]) # sys.argv[1] file path to csv
    survey_df.show()

    logger.info("Finished First Spark App")
    spark.stop()
```

O/P

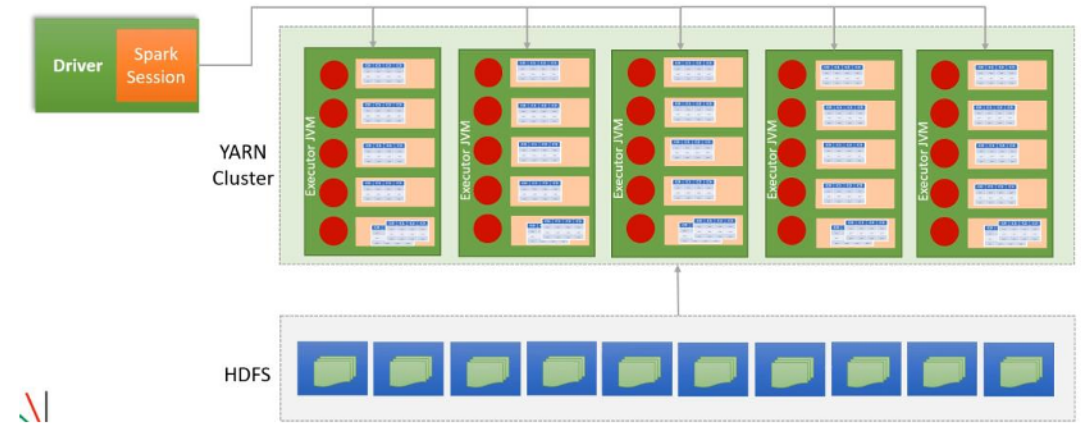
```
23/02/09 23:59:06 INFO Hello Spark: Starting First Spark App
23/02/09 23:59:16 WARN package: Truncated the string representati
+-------------------+---+------+--------------+-----+-----------
|          Timestamp|Age|Gender|       Country|state|self_employe
+-------------------+---+------+--------------+-----+-----------
|2014-08-27 11:29:31| 37|Female| United States|   IL|          N
|2014-08-27 11:29:37| 44|     M| United States|   IN|          N
|2014-08-27 11:29:44| 32|  Male|        Canada|   NA|          N
|2014-08-27 11:29:46| 31|  Male|United Kingdom|   NA|          N
|2014-08-27 11:30:22| 31|  Male| United States|   TX|          N
|2014-08-27 11:31:22| 33|  Male| United States|   TN|          N
|2014-08-27 11:31:50| 35|Female| United States|   MI|          N
```

- **Data Frame Partitions and Executors**
  - Spark data frame is distributed data structure and how it help to Spark to implement in precessing.



dataframe partitions are stored distributed storage nodes on HDFS or clous amazon S3

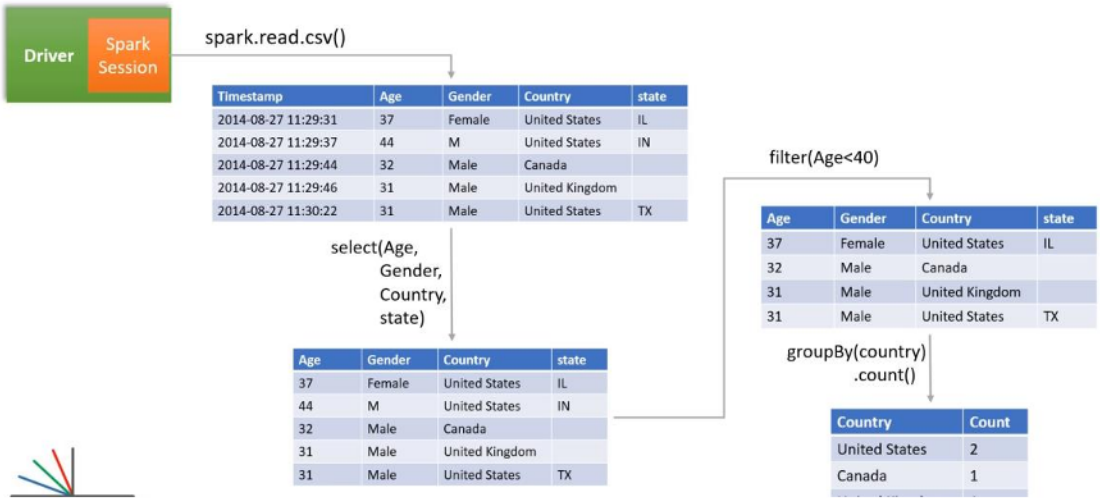
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv Submit Config vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv



- **Spark Transformations and Action**
  - Spark Data Frame is immutable data structure.

Spark Transformations?

  - You can give instruction to driver what do you want to do.
  - Driver decides it how to achieve the instructions by executors these instructions are called as Transformations. These Transformations are same like SQL operations.

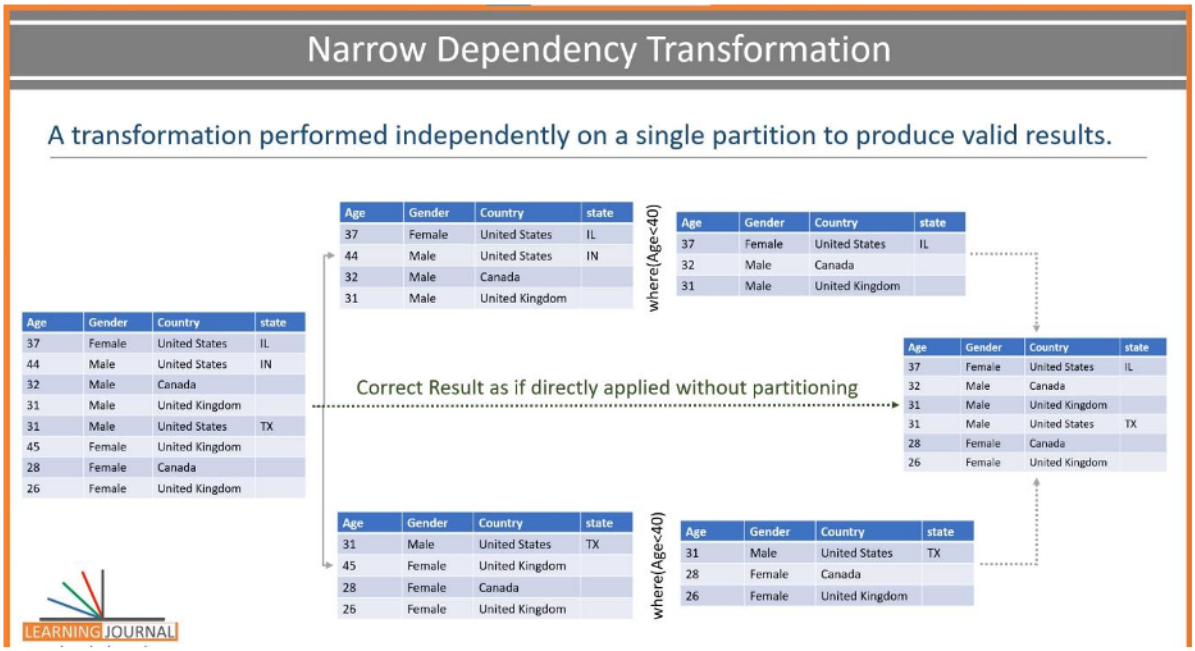


1. Transformations

1. Narrow Dependency
2. Wide Dependency

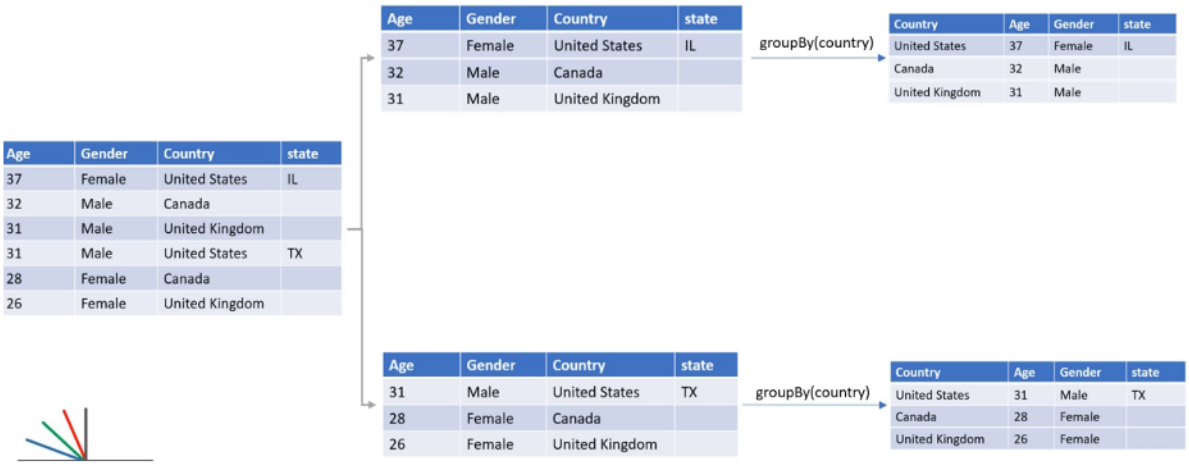
Narrow Vs. Wide Transformations?

"Narrow"

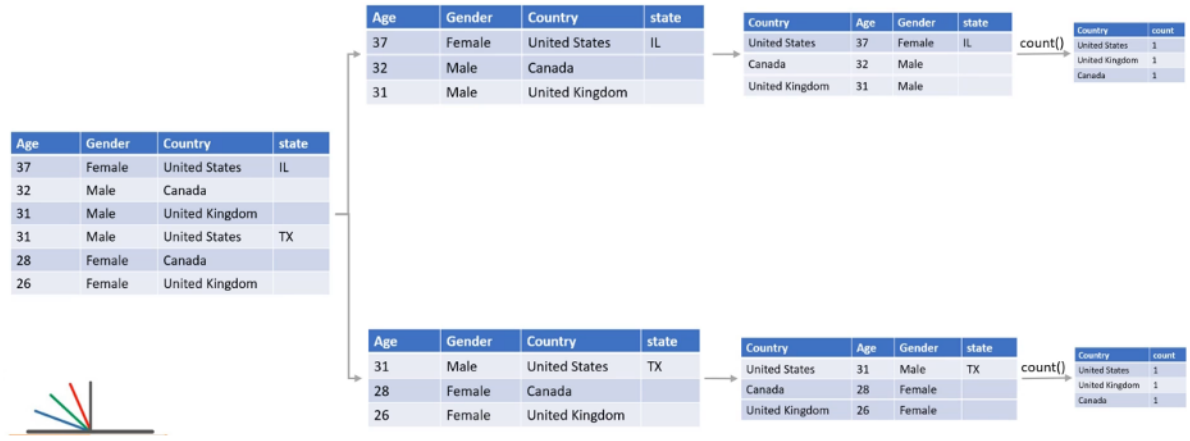


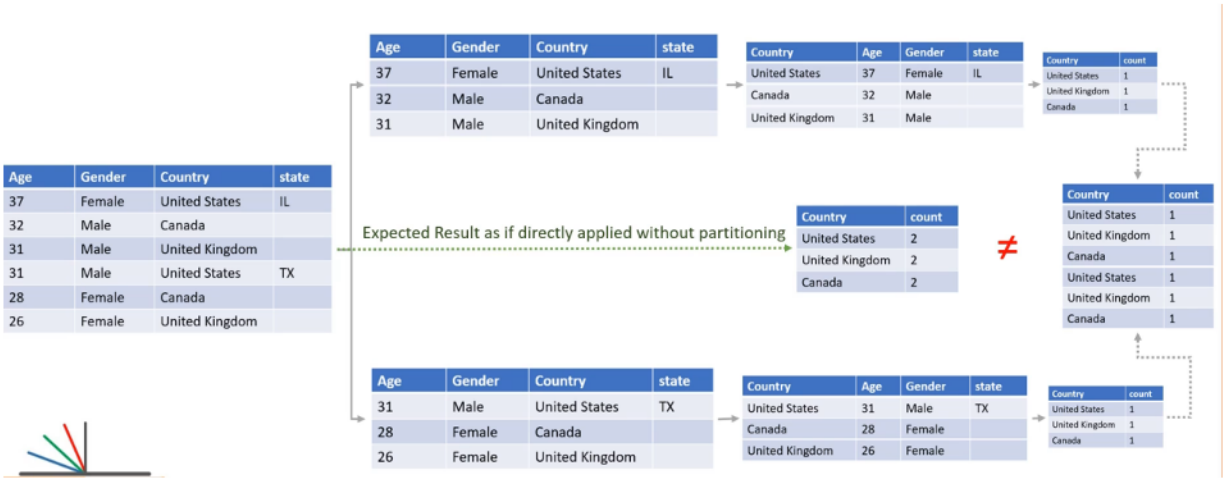
"Wide"

A transformation that requires data from other partitions to produce valid results.



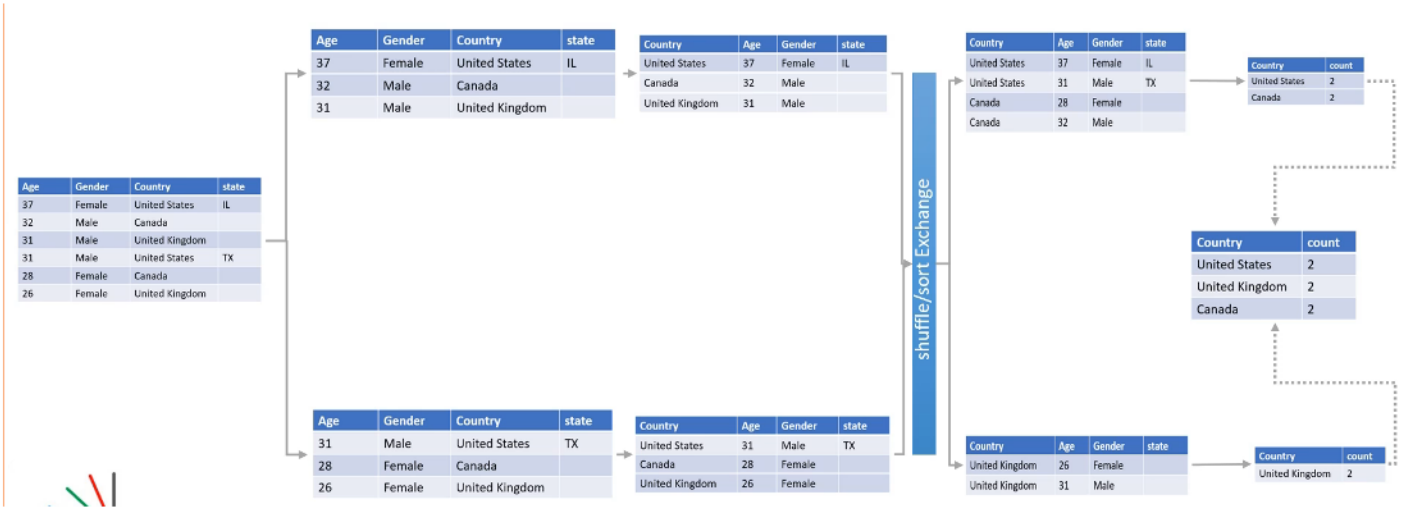
Group By breaks the independence of the partition





This kind of transformation will create incorrect results how we can correct it? Combine & Re-partition data are known as Shuffle / sort Exchange operations.

Group by, Join, Order By, Distinct, etc. are wide dependency transformations.



Lazy Evaluations?

Its functional programming technique.

```
spark = SparkSession \
    .builder \
    .config(conf=conf) \
    .getOrCreate()

survey_df = load_survey_df(spark, sys.argv[1])
filtered_df = survey_df.where("Age < 40")
selected_df = filtered_df.select("Age", "Gender", "Country", "state")
grouped_df = selected_df.groupBy("Country")
count_df = grouped_df.count()

count_df.show()
```

Typical program runs line by line. But Spark Programs are not behaves same way because in Spark builder pattern is used i.e. DAG of transformations.

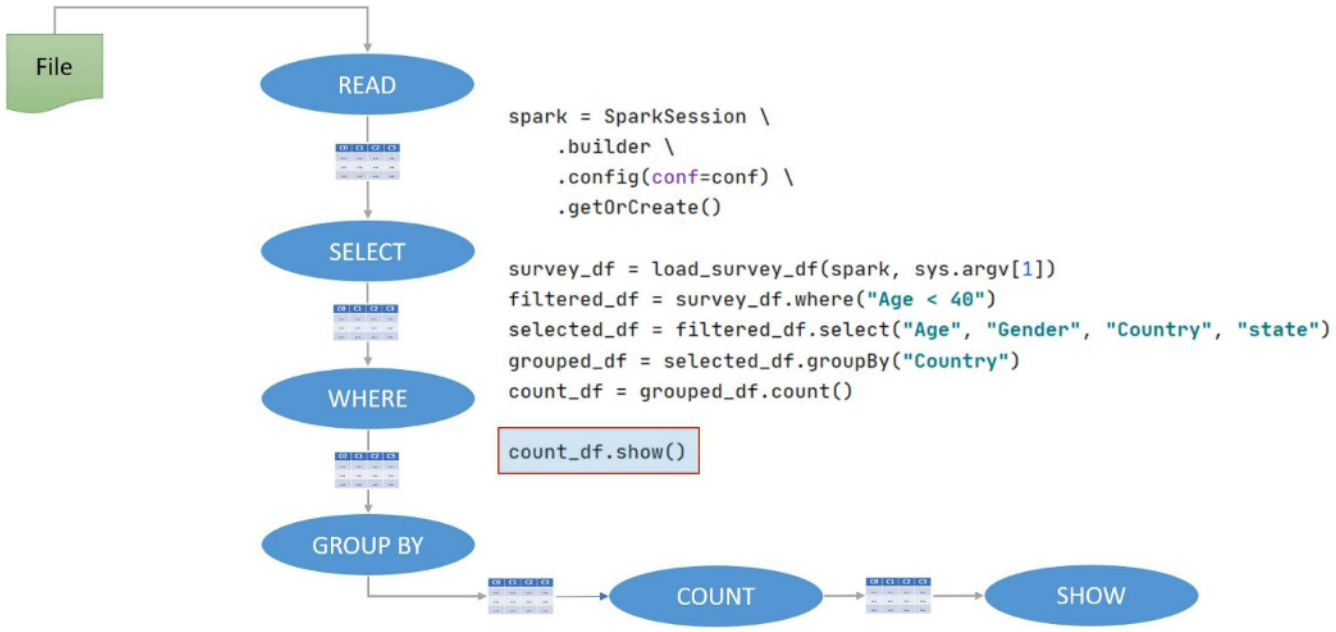
```
spark = SparkSession \
    .builder \
    .config(conf=conf) \
    .getOrCreate()
```

```
survey_df = load_survey_df(spark, sys.argv[1])
filtered_df = survey_df.where("Age < 40")
selected_df = filtered_df.select("Age", "Gender", "Country", "state")
grouped_df = selected_df.groupBy("Country")
count_df = grouped_df.count()

count_df.show()
```

All of these go to the spark driver and rearrange them for optimized certain activities and finally create an execution plan which will be executed by the executors which is terminated by and triggered by Action.

- What's an action? READ, WRITE, COLLECT and SHOW all these are actions.

Spark Actions



Show( ) is an action (actions terminates the trasformation DAG and trigger the Execution)

- Trasformations are lazy and actions are evaluate immediately.

```
- import sys
  from pyspark.sql import *
  from lib.utils import get_spark_app_config, load_survey_df
  from lib.logger import Log4j

  if __name__ == "__main__":
      conf = get_spark_app_config()
      spark = SparkSession.builder \
          .config(conf=conf)\
          .getOrCreate() \

      logger = Log4j(spark)

      if len(sys.argv) !=2: # Command line argument if not provided we get error
          logger.error("Usage: HelloSpark <filename>")
          sys.exit(-1)

      logger.info("Starting First Spark App")
      # Your processing code
```

```
        survey_df = load_survey_df(spark, sys.argv[1]) # Schema and DF

        count_df = survey_df\
            .where("Age < 40")\
            .select("Age","Gender", "Country", "state") \
            .groupby("Country") \
            .count() # count_df Change of transformation

        count_df.show() # Action

        logger.info("Finished First Spark App")
        spark.stop()
```

``

```
23/02/10 11:21:53 INFO Hello Spark: Starting First Spark App
+--------------+-----+
|       Country|count|
+--------------+-----+
| United States|    4|
|        Canada|    2|
|United Kingdom|    1|
+--------------+-----+

23/02/10 11:22:01 INFO Hello Spark: Finished First Spark App
```

- by creating function

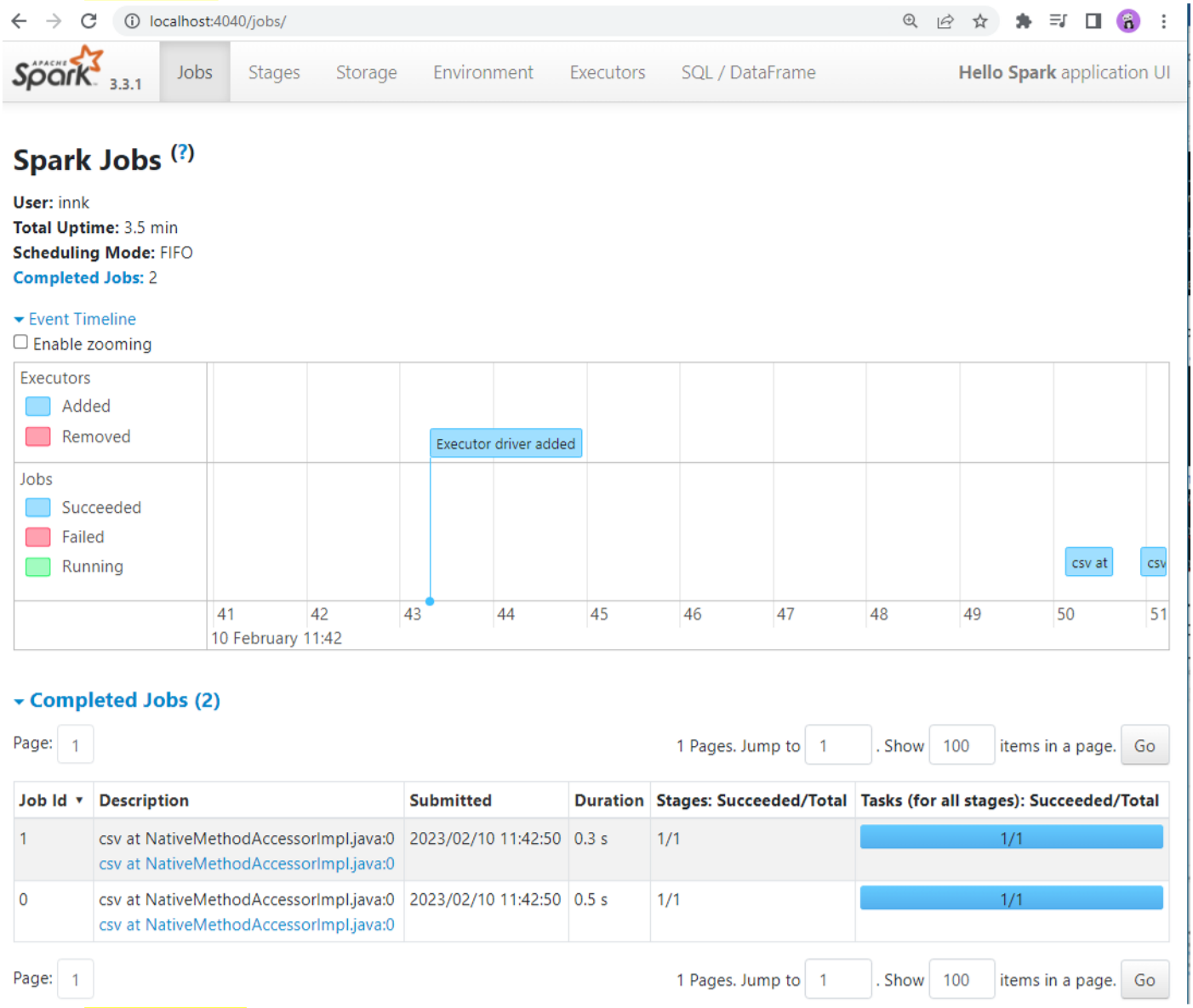
utils.py

```
- def count_by_country(survey_df):
        return survey_df \
            .where("Age < 40") \
            .select("Age", "Gender", "Country", "state") \
            .groupby("Country") \
            .count()
```

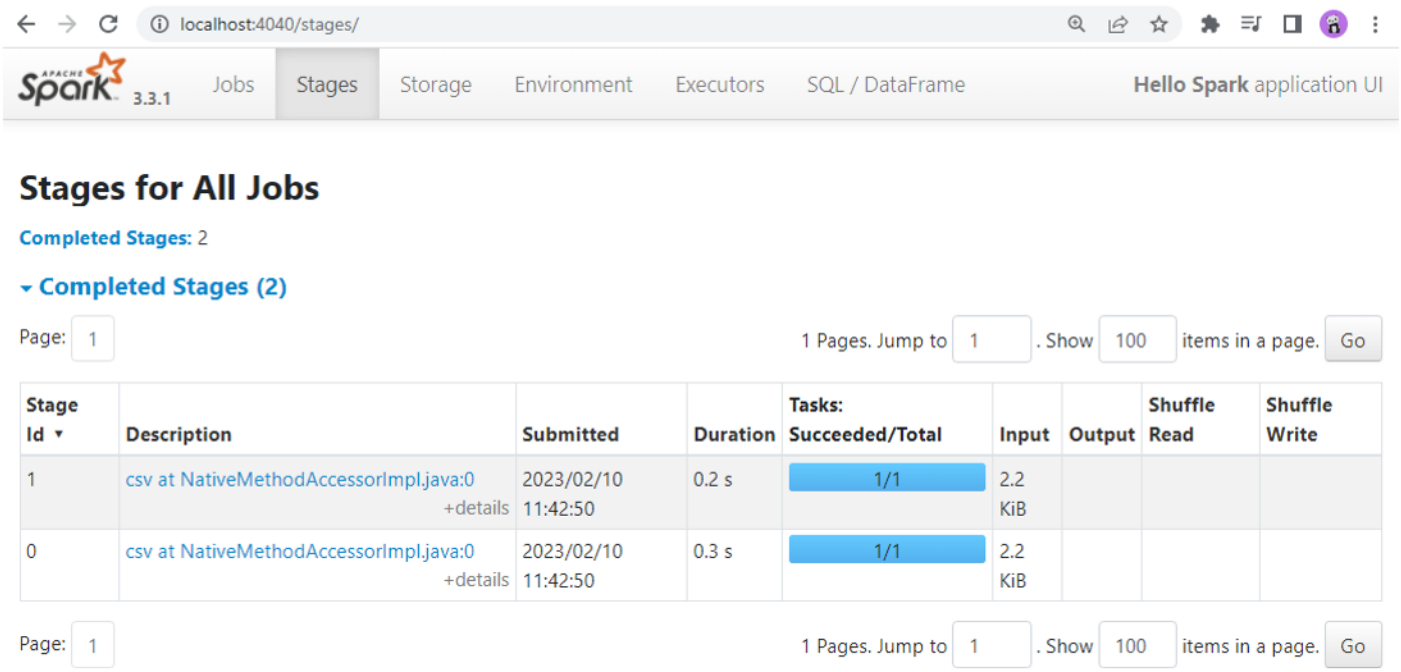
main.py

```
survey_df = load_survey_df(spark, sys.argv[1]) # Schema and DF

count_df = count_by_country(survey_df)

count_df.show() # Action
```

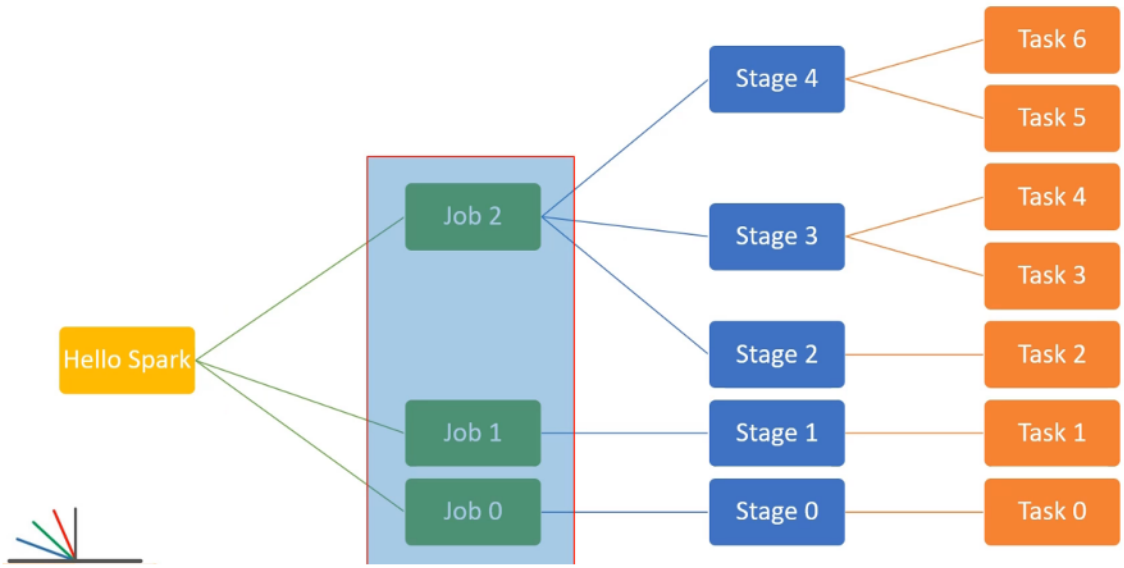
- **Spark Jobs, Stages and Task ref. 23**
  - Controlling Internal operations
  - UI - Jobs



  - UI – Stages



Spark Execution Plan



main.py

```
import sys
from pyspark.sql import *
from lib.utils import get_spark_app_config, load_survey_df, count_by_country
from lib.logger import Log4j

if __name__ == "__main__":
    conf = get_spark_app_config()
    spark = SparkSession.builder \
        .config(conf=conf)\
        .getOrCreate() \

    logger = Log4j(spark)

    if len(sys.argv) !=2: # Command line argument if not provided we get error
        logger.error("Usage: HelloSpark <filename>")
        sys.exit(-1)

    logger.info("Starting First Spark App")
    # Your processing code

    survey_df = load_survey_df(spark, sys.argv[1]) # Schema and DF

    partitioned_survey_df = survey_df.repartition(2) # It takes DF as I/p and produces
another DF this new DF should have 2 partitions

    count_df = count_by_country(partitioned_survey_df) # We are using this partitions
for rest of the transformations.

    input("Press Enter") # for local debugging in UI to hold program to check in UI

    logger.info(count_df.collect()) # Collect Action | The collect action returns the
DF as Python List
    # show is used for printing, debugging and show method compiles down to a complex
internal code it creates
    # unnecessary confusion
    logger.info("Finished First Spark App")
    spark.stop()
```

spark.conf

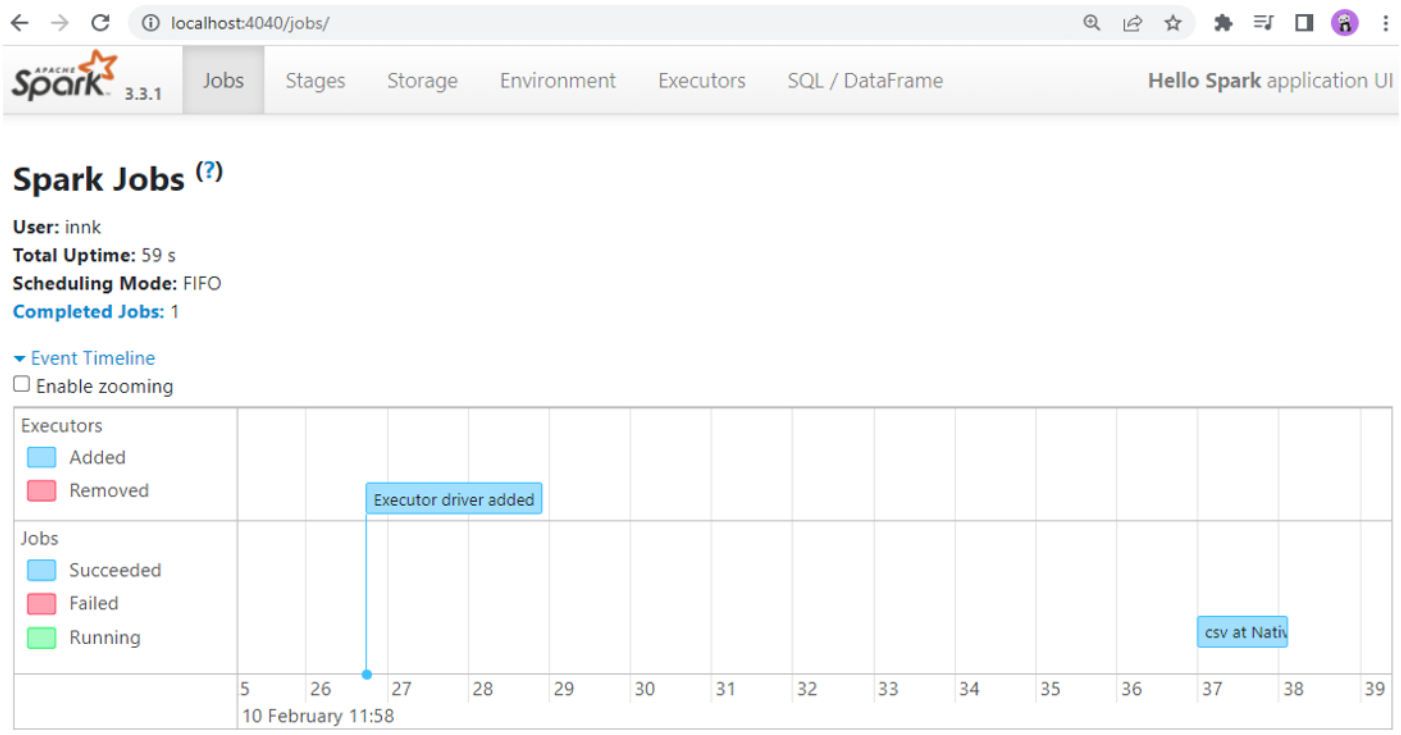
```
[SPARK_APP_CONFIGS]
spark.app.name = Hello Spark
spark.master = local[3]
spark.sql.shuffle.partitions = 2
# its used for the controlling partitions because in internal operations
# we dont know how many partitions are made so we need to control it.
```

utils.py

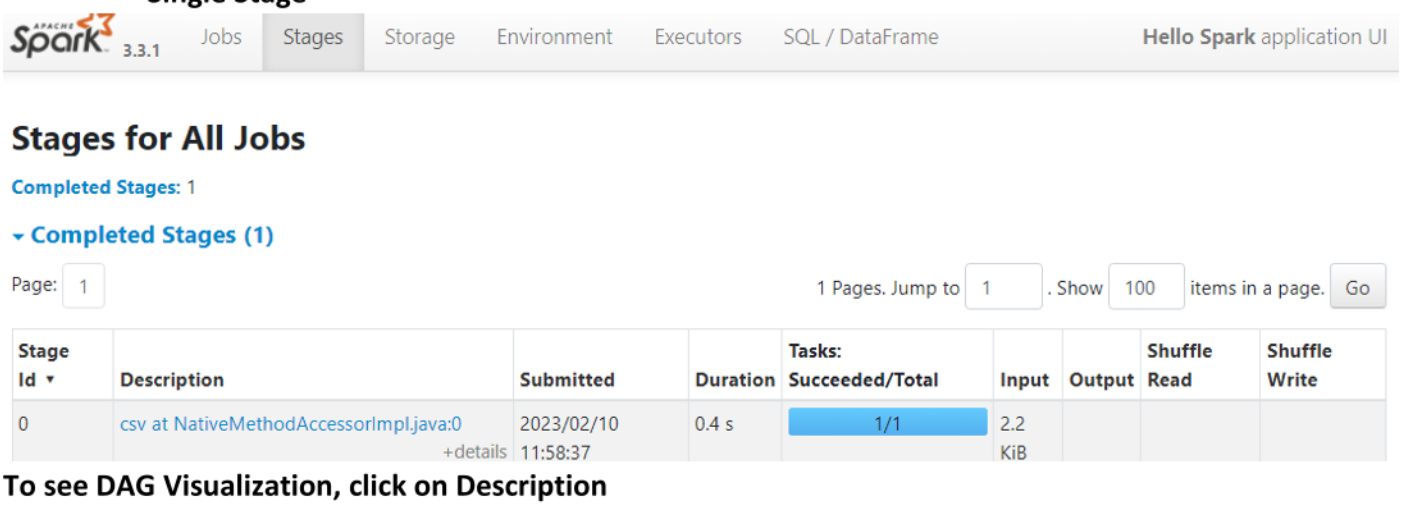
```
def count_by_country(survey_df):
    return survey_df \
        .where("Age < 40") \
        .select("Age", "Gender", "Country", "state") \
        .groupby("Country") \
        .count()
```

- **Understanding Execution Plan**
  - Every Spark action is transmitted into a JOB as follow
  - Loading only single csv file created single action that creates single action as follow

```
survey_df = load_survey_df(spark, sys.argv[1]) # now in this code we are just loading data into DF only.
input("Press Enter")
```



- **Single Stage**

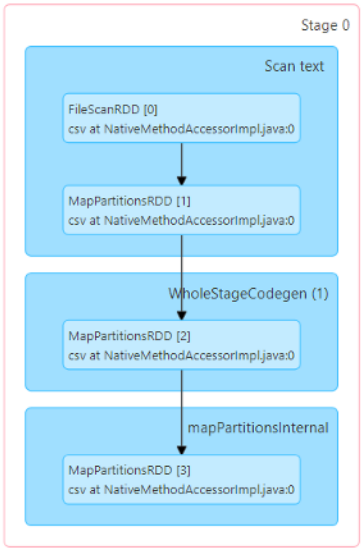


To see DAG Visualization, click on Description

Details for Stage 0 (Attempt 0)

Resource Profile Id: 0
Total Time Across All Tasks: 0.2 s
Locality Level Summary: Process local: 1
Input Size / Records: 2.2 KiB / 1
Associated Job Ids: 0

▾ DAG Visualization



▸ Show Additional Metrics
▸ Event Timeline

- DAG shows internal processes the sequence is compiled code which is generated by the spark.

Spark Execution Plan

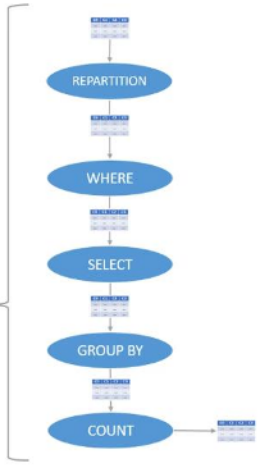
```
survey_df = spark.read \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .csv(sys.argv[1])
```

Job 0

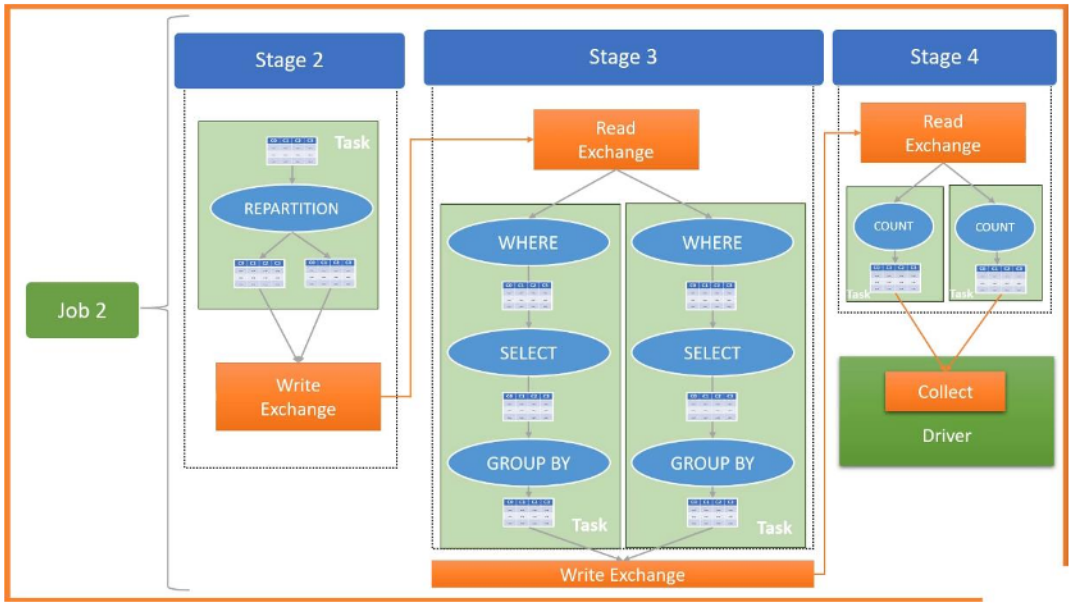
Job 1

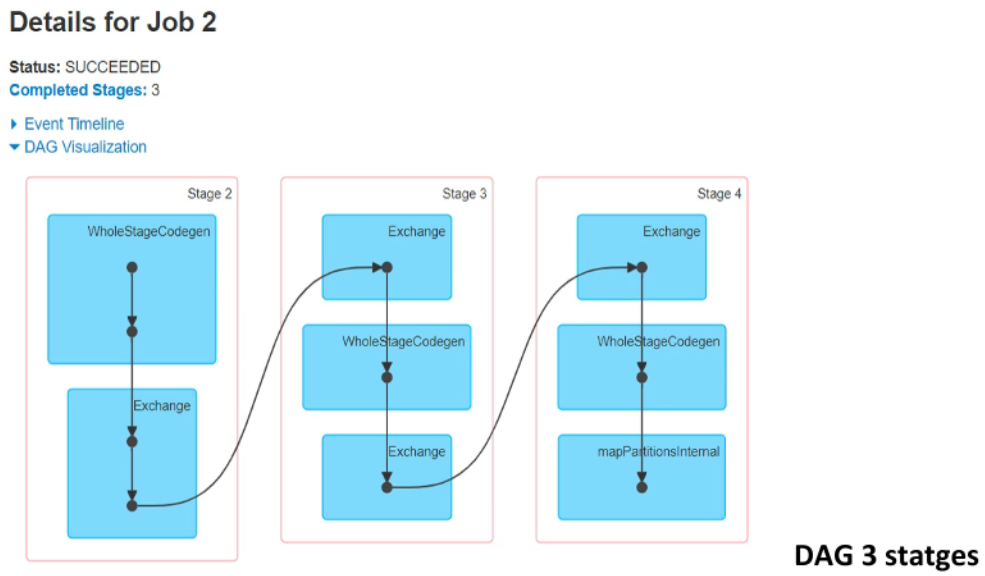
```
partitioned_survey_df = survey_raw_df.repartition(2)
count_df = survey_df.filter("Age < 40") \
    .select("Age", "Gender", "Country", "state") \
    .groupBy("Country") \
    .count()
logger.info(count_df.collect())
```

Job 2



Shuffle/sort Exchange





DAG 3 statges

```
23/02/10 12:30:01 INFO Hello Spark: Starting First Spark App
23/02/10 12:30:12 INFO Hello Spark: [[Canada, 2], [United States, 4], [United Kingdom, 1]]
23/02/10 12:34:16 INFO Hello Spark: Finished First Spark App
```

Log with count list

- **Unit Testing Spark Application**

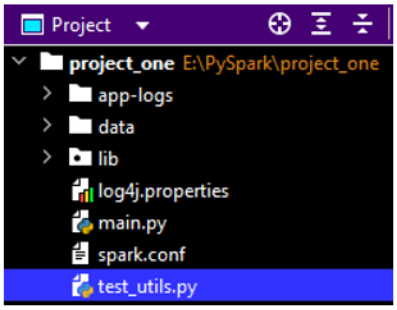
Test Case: Make sure that we are loading the data file correctly.

S. No.	Test Step	Test Data
1.	Read the data file	sample.csv
2.	Validate the number of records	record count = 9

Test Case: Record Count by country is computed correctly

S. No.	Test Step	Test Data
1.	Read the data file	sample.csv
2.	Validate count by country	United Kingdom = 1 Canada = 2 United States = 4

>>



test_utils.py

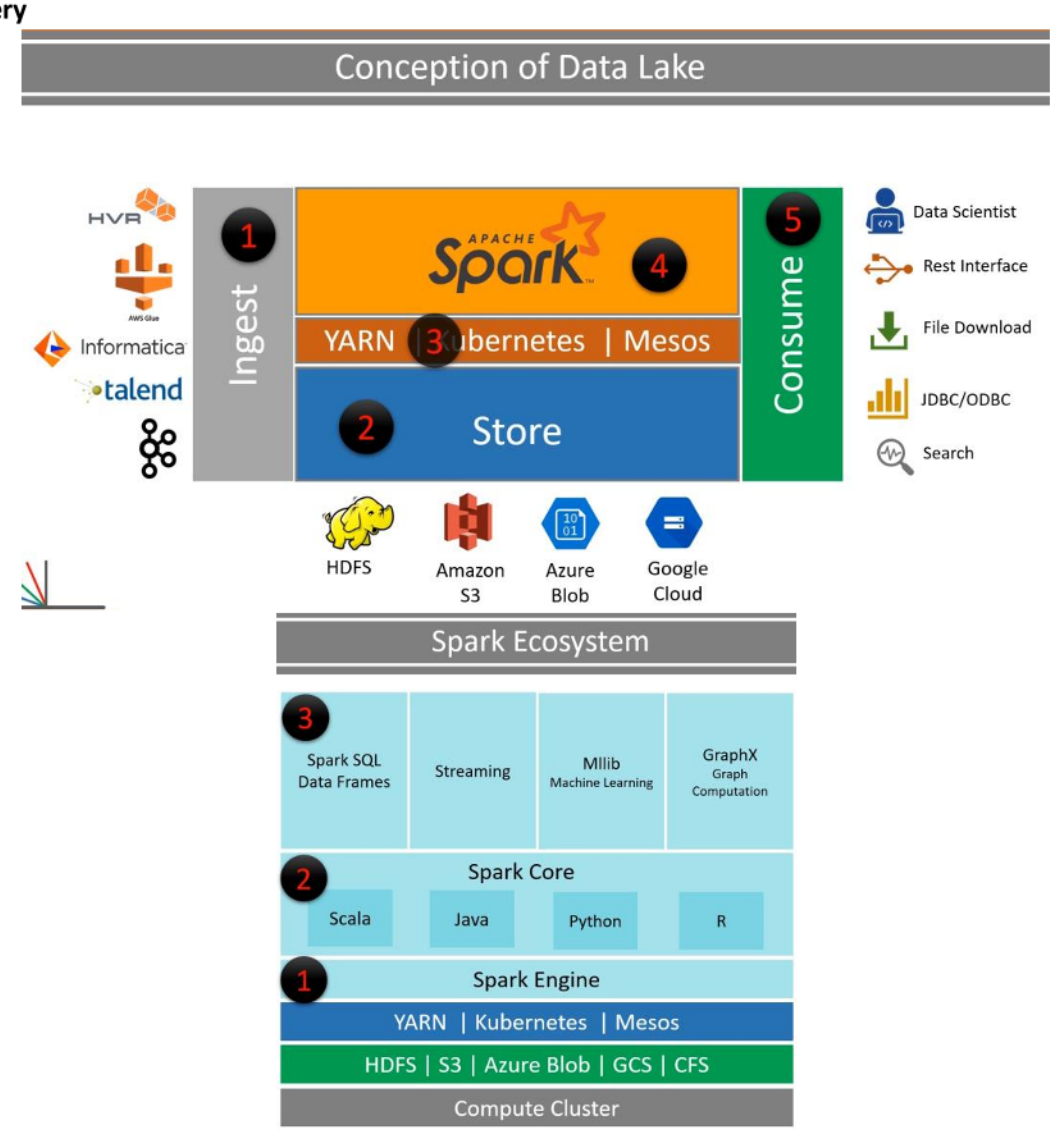
```
from unittest import TestCase
from pyspark.sql import SparkSession
from lib.utils import load_survey_df,count_by_country

class UtilsTestCase(TestCase):

    @classmethod
    def setUpClass(cls) -> None:
        cls.spark = SparkSession.builder \
                .master("local[3]") \
                .appName("SparkTest") \
                .getOrCreate()

    def test_datafile_loading(self):
        sample_df = load_survey_df(self.spark, "data/sample.csv")
        result_count = sample_df.count()
        self.assertEqual(result_count, 9, "Record count should be 9")

    def test_country_count(self):
        sample_df = load_survey_df(self.spark, "data/sample.csv")
        count_list = count_by_country(sample_df).collect()
        count_dict = dict()
        for row in count_list:
            count_dict[row["Country"]] = row["count"]
        self.assertEqual(count_dict["United States"], 4, "Count for United State should be 4")
        self.assertEqual(count_dict["Canada"], 2, "Count for United State should be 2")
        self.assertEqual(count_dict["United Kingdom"], 1, "Count for United State should be 1")

    # @classmethod
    # def tearDownClass(cls) -> None:
    #     cls.spark.stop()
```

```
Run:  Python tests for test_utils.UtilsTestCase
Tests passed: 2 of 2 tests – 6 sec 817 ms
Test Results  6 sec 817 ms
C:\Users\innk\anaconda3\envs\project_one\python.exe "C:/Program Files/JetBrains/PyCharm Communi
Testing started at 12:57 pm ...
Launching pytest with arguments test_utils.py::UtilsTestCase --no-header --no-summary -q in E:\

============================= test session starts =============================
collecting ... collected 2 items

test_utils.py::UtilsTestCase::test_country_count
test_utils.py::UtilsTestCase::test_datafile_loading

============================== 2 passed in 13.14s ==============================

Process finished with exit code 0
PASSED                  [ 50%]{'United States': 4, 'Canada': 2, 'United Kingdom': 1}
PASSED                [100%]
```

```
+--------------+-----+
|       Country|count|
+--------------+-----+
| United States|    4|
|        Canada|    2|
|United Kingdom|    1|
+--------------+-----+
```

count table for ref.

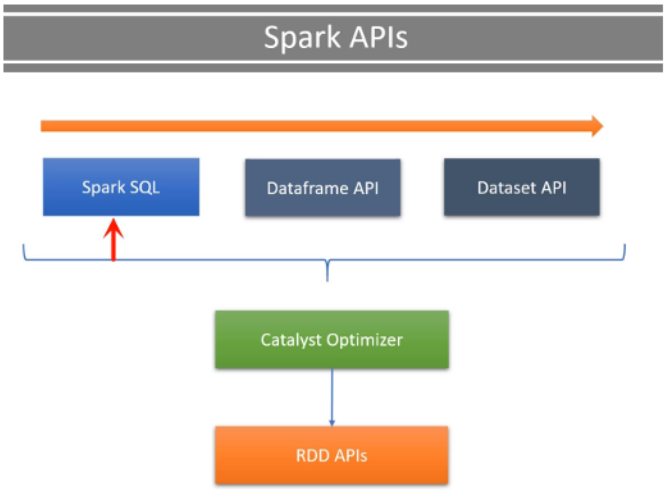
- **Summery**



Developer Experience

1. Creating and Configuring Spark Project using your IDE
2. Configuring Log4J for your Spark Application
3. Creating and Configuring Spark Session
4. Managing your Spark Session Configurations using spark.conf
5. Creating a modular Structure for your Spark Application
6. Unit Testing Spark Application
7. Building and packaging your Spark Application
8. Deploying your Spark Application on a Cluster
9. Collecting Application Logs from Spark Cluster

- **Spark Structured API Foundation**



- Spark SQL: Most convenient to use.

- DF API**: lesser than Spark SQL. Any favourite language is suitable.

- Dataset API: Lesser than above both. Language native such as Skala, java (which are JVM based languages)

- Catalyst Optimizer: SparkSQL, DF API, Dataset API. Optimizer decides how the code is executed and execution plan

- RDD not recommended by spark

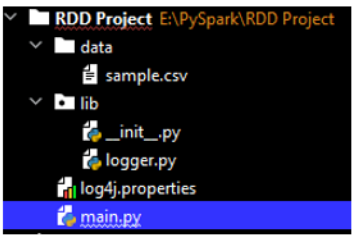
- **Spark RDD APIs (Resilient Distributed Dataset)** ref. 36

How to Create RDD?

Creating Data frame

1. Create a SparkConf object
2. Create a Spark Session using your SparkConf
3. Use the spark session to read the data file

```
conf = SparkConf()
conf.setMaster("local[3]").setAppName("HelloRDD")

spark = SparkSession.builder.config(conf=conf).getOrCreate()

surveyDF = spark.read.option("header", "true").option("inferSchema", "true")
```

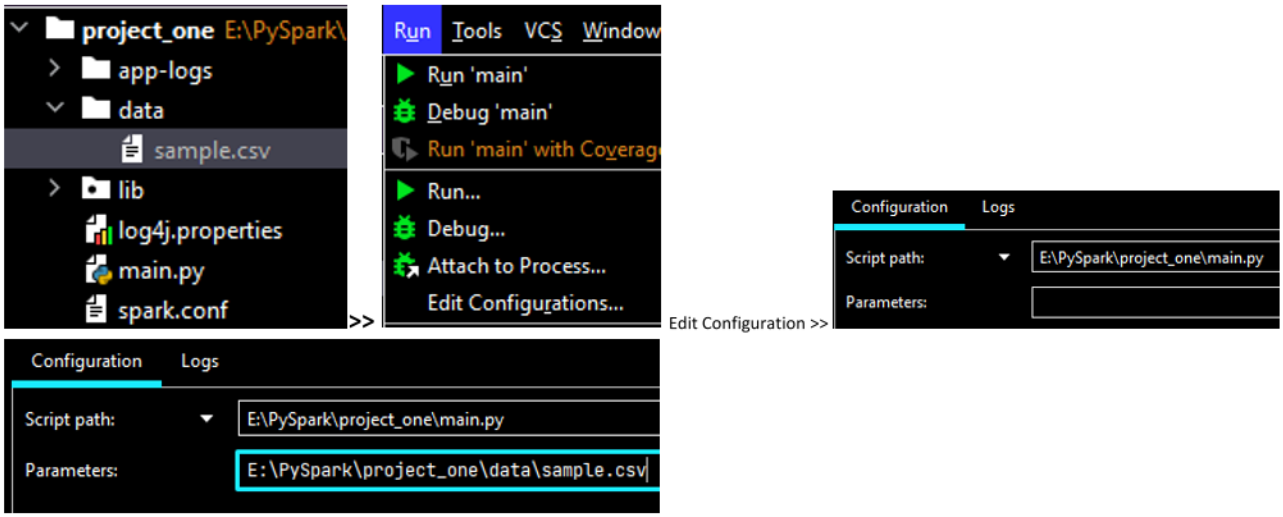


main.py

```
import sys
from collections import namedtuple
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession
from lib.logger import Log4j

SurveyRecord = namedtuple("SurveyRecord", ["Age", "Gender", "Country",
"State"])

if __name__ == "__main__":
    conf = SparkConf() \
        .setMaster("local[3]") \
        .setAppName("RDDProject")

    # sc = SparkContext(conf=conf)
    # alternet method to get spark context

    spark = SparkSession \
        .builder \
        .config(conf=conf) \
        .getOrCreate()

    sc = spark.sparkContext
    logger = Log4j(spark)

    if len(sys.argv) != 2:
```

```
        logger.error("Usage: HelloSpark <filename>")
        sys.exit(-1)

    lineRDD = sc.textFile(sys.argv[1])
    partitionedRDD = lineRDD.repartition(2)

    colsRDD = partitionedRDD.map(lambda line: line.replace('"', '').split(","))
# removing " from csv then o/p list of text.
    selectRDD = colsRDD.map(lambda cols: SurveyRecord(int(cols[1]), cols[2],
cols[3], cols[4]))  # Attaching schema to RDD with help of SurveyRecord
    filteredRDD = selectRDD.filter(lambda r: r.Age < 40)
    kvRDD = filteredRDD.map(lambda r: (r.Country, 1))  # kv : key value where
Country becomes key and 1 hardcoded value
    countRDD = kvRDD.reduceByKey(lambda v1, v2: v1 + v2)

    colsList = countRDD.collect()
    for x in colsList:
        logger.info(x)
```

- **Run / Debug Configuration**



*****csv file path if this parameter is not config then code not able to find csv with sys.argv[1]*****

```
"C:\Users\innk\anaconda3\envs\RDD Project\python.exe"
data/sample.csv
E:\pySpark_soft\spark_3\python\lib\pyspark.zip\pyspar
E:\pySpark_soft\spark_3\python\lib\pyspark.zip\pyspar
E:\pySpark_soft\spark_3\python\lib\pyspark.zip\pyspar
E:\pySpark_soft\spark_3\python\lib\pyspark.zip\pyspar
23/02/13 13:41:40 INFO HelloRDD: [United States, 4]
23/02/13 13:41:40 INFO HelloRDD: [Canada, 2]
23/02/13 13:41:40 INFO HelloRDD: [United Kingdom, 1]
```

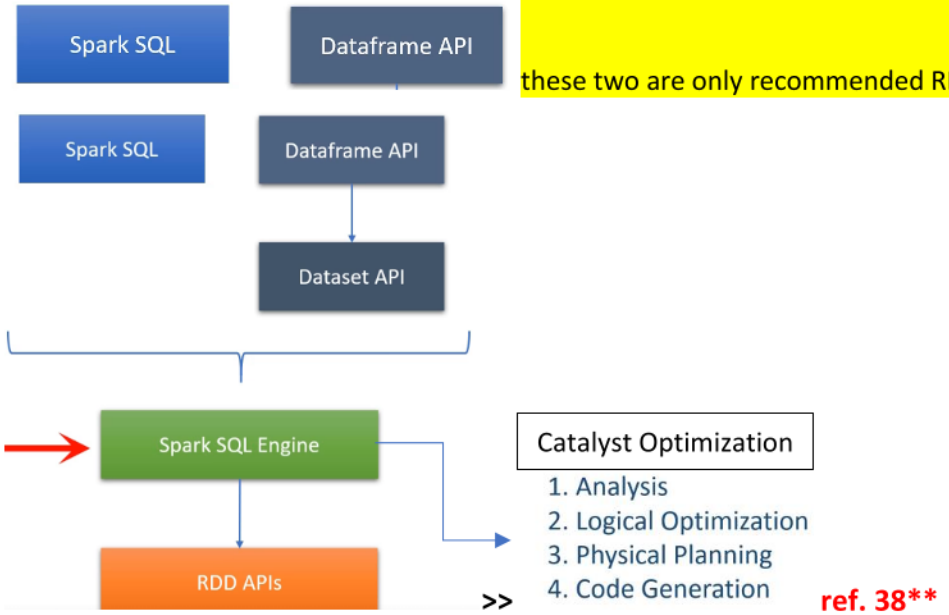
- **Working with Spark SQL**

main.py

```
import sys

from pyspark.sql import SparkSession

from lib.logger import Log4j


if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .master("local[3]") \
        .appName("SparkSQL") \
        .getOrCreate()

    logger = Log4j(spark)

    if len(sys.argv) != 2:
        logger.error("Usage: HelloRDD <filename>")
        sys.exit(-1)

    surveyDF = spark.read \
        .option("header", "true") \
        .option("inferSchema", "true") \
        .csv(sys.argv[1])

    # Spark allows you to register data frame as a view
    surveyDF.createOrReplaceTempView("survey_tbl") #View
    countDF = spark.sql("SELECT Country, COUNT(1) AS COUNT FROM survey_tbl WHERE Age<40
GROUP BY Country") # SQL code
    countDF.show()
```

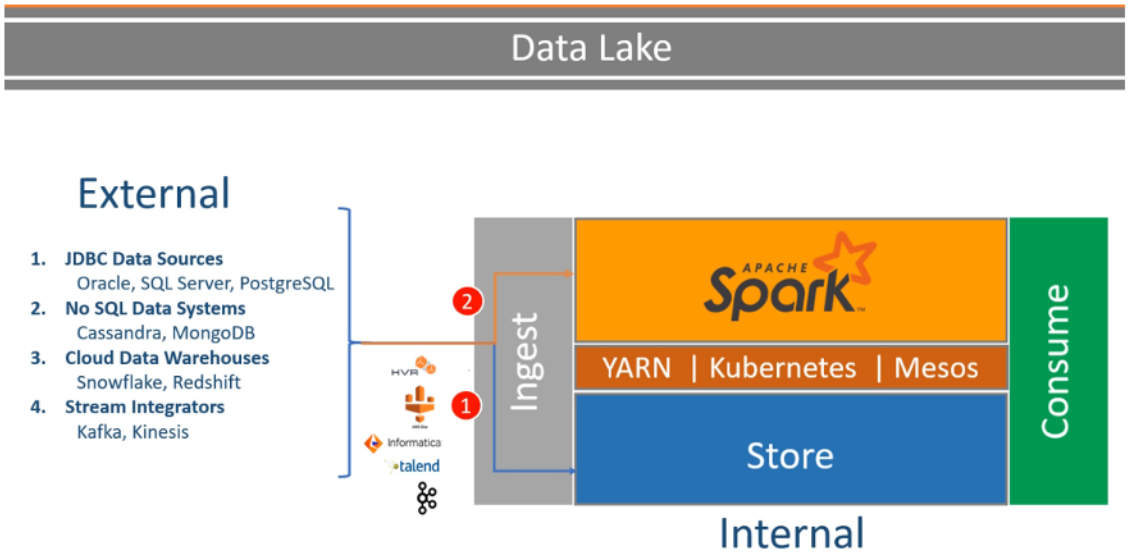
```
+--------------+-----+
|       Country|COUNT|
+--------------+-----+
| United States|    4|
|        Canada|    2|
|United Kingdom|    1|
+--------------+-----+
```

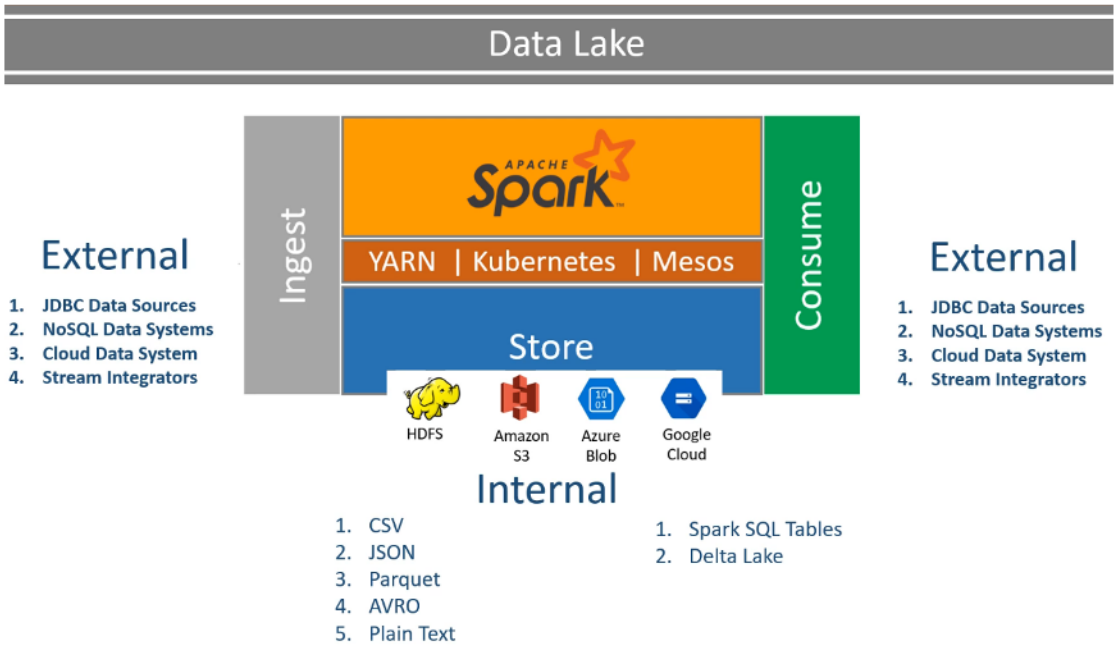
- **Spark SQL Engine and Catalyst Optimizer**



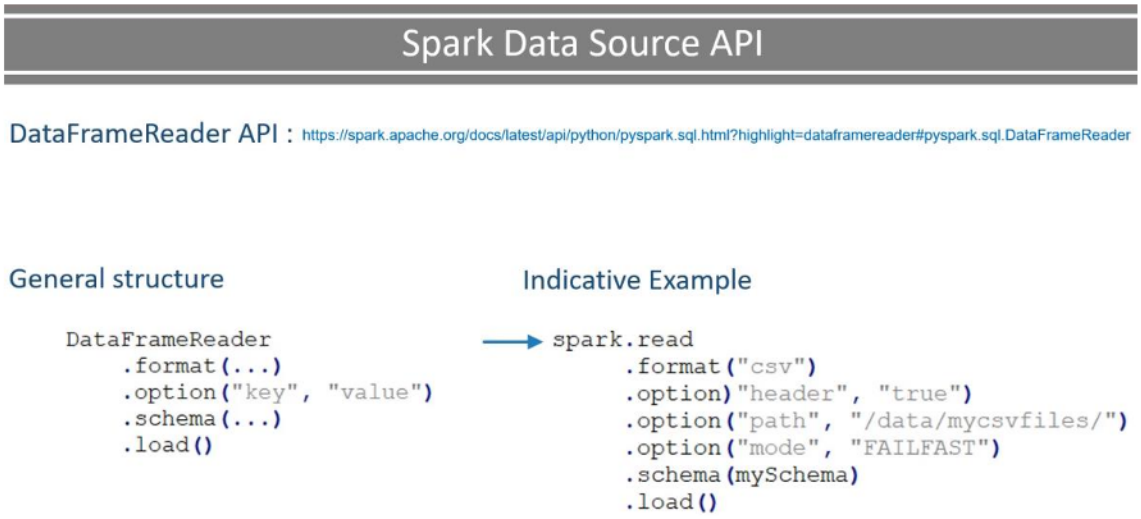
ref. 38**

- **Spark Data Sources and Sinks**





- **Spark Data Frame Reader API**



Indicative Example

```
spark.read
     .format("csv")
     .option("header", "true")
     .option("path", "/data/mycsvfiles/")
     .option("mode", "FAILFAST")
     .schema(mySchema)
     .load()
```

Community Formats
Cassandra, MongoDB, AVRO, XML, HBase, Redshift

```
.schema(mySchema)
.load()
```

Schema
1. Explicit
2. Infer Schema
3. Implicit

Indicative Example

```
spark.read
     .format("csv")
     .option("header", "true")
     .option("path", "/data/mycsvfiles/")
     .option("mode", "FAILFAST")
     .schema(mySchema)
     .load()
```

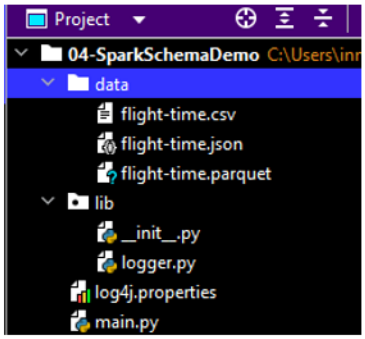
Read Mode
1. PERMISSIVE
2. DROPMALFORMED
3. FAILFAST

1. **Permissive** mode is default option. This set all the field to the null when it encounters a corrupted record. Places that record is string column called _currupt_record
2. **DropMalFormed** mode remove the malformed record. Ignoring the malformed records only loading well formed records only.
3. **FAILFAST** raises and exceptions and terminate when detection of mal formed record.

Its standard code. Don't use shortcuts.

- **Reading CSV, JSON, and Parquet files.**

1. How to use DataFrameReader for CSV, JSON, and Parquet



main.py (without schema)

```
from pyspark.sql import SparkSession

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .master("local[3]") \
        .appName("spark_schema_demo") \
        .getOrCreate()

    logger = Log4j(spark)

    flight_time_csv = spark.read \
        .format("csv")\
        .option("header", "true")\
        .load("data/flight*.csv")

    flight_time_csv.show(5)
    logger.info("CSV Schema: " + flight_time_csv.schema.simpleString())
```

```
+--------+----------+----------------+------+----
| FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|ORIG
+--------+----------+----------------+------+----
|1/1/2000|        DL|            1451|   BOS|
|1/1/2000|        DL|            1479|   BOS|
|1/1/2000|        DL|            1857|   BOS|
|1/1/2000|        DL|            1997|   BOS|
|1/1/2000|        DL|            2065|   BOS|
+--------+----------+----------------+------+----
only showing top 5 rows
```

Result (int result as string here)

INFO spark_schema_demo: CSV Schema:
struct<FL_DATE:string,OP_CARRIER:string,OP_CARRIER_FL_NUM:string,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:string,DEP_TIME:string,WHEELS_ON:string,TAXI_IN:string,CRS_ARR_TIME:string,ARR_TIME:string,CANCELLED:string,DISTANCE:string>

main.py (with schema) in this case all numerical are result as integers.

```
flight_time_csv = spark.read \
    .format("csv")\
    .option("header", "true")\
    .option("inferSchema", "true")\
    .load("data/flight*.csv")
```

Just adding Schema

INFO spark_schema_demo: CSV Schema:
struct<FL_DATE:string,OP_CARRIER:string,OP_CARRIER_FL_NUM:int,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:int,DEP_TIME:int,WHEELS_ON:int,TAXI_IN:int,CRS_ARR_TIME:int,ARR_TIME:int,CANCELLED:int,DISTANCE:int>

- You can't just be relied on inferScheam only. Because in result we got Date as string in above output.

Schema for DataFrame

1. Explicit
2. Implicit

main.py (for JSON)

```
flight_time_json_df = spark.read \
    .format("json") \
    .load("data/flight*.json")

flight_time_json_df.show(5)
logger.info("JSON Schema: " + flight_time_json_df.schema.simpleString())
```

```
+--------+---------+------------+------------+--------+----+-------------+--------+-----
|ARR_TIME|CANCELLED|CRS_ARR_TIME|CRS_DEP_TIME|DEP_TIME|DEST|DEST_CITY_NAME|DISTANCE| FL_D
+--------+---------+------------+------------+--------+----+-------------+--------+-----
|    1348|        0|        1400|        1115|    1113| ATL|   Atlanta, GA|     946|1/1/2
|    1543|        0|        1559|        1315|    1311| ATL|   Atlanta, GA|     946|1/1/2
|    1651|        0|        1721|        1415|    1414| ATL|   Atlanta, GA|     946|1/1/2
|    2005|        0|        2013|        1715|    1720| ATL|   Atlanta, GA|     946|1/1/2
|    2240|        0|        2300|        2015|    2010| ATL|   Atlanta, GA|     946|1/1/2
+--------+---------+------------+------------+--------+----+-------------+--------+-----
only showing top 5 rows

23/02/26 13:48:23 INFO spark_schema_demo: JSON Schema: struct<ARR_TIME:bigint,CANCELLED:b
```

DF results in alphabetical format.

INFO spark_schema_demo: JSON Schema:
struct<ARR_TIME:bigint,CANCELLED:bigint,CRS_ARR_TIME:bigint,CRS_DEP_TIME:bigint,DEP_TIME:bigint,DEST:string,DEST_CITY_NAME:string,DISTANCE:bigint,FL_DATE:string,OP_CARRIER:string,OP_CARRIER_FL_NUM:bigint,ORIGIN:string,ORIGIN_CITY_NAME:string,TAXI_IN:bigint,WHEELS_ON:bigint>

Date still in string.

Schema for DataFrame

1. Explicit
2. Implicit

main.py (for parquet format)

```
flight_time_parquet_df = spark.read \
      .format("parquet") \
      .load("data/flight*.parquet")

   flight_time_parquet_df.show(5)
   logger.info("PARQUET Schema: " + flight_time_parquet_df.schema.simpleString())
```

```
+----------+----------+-----------------+------+---------------+----+-------------+------------+
|   FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|ORIGIN_CITY_NAME|DEST|DEST_CITY_NAME|CRS_DEP_TIME|
+----------+----------+-----------------+------+---------------+----+-------------+------------+
|2000-01-01|        DL|             1451|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1115|
|2000-01-01|        DL|             1479|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1315|
|2000-01-01|        DL|             1857|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1415|
|2000-01-01|        DL|             1997|   BOS|      Boston, MA| ATL|   Atlanta, GA|        1715|
|2000-01-01|        DL|             2065|   BOS|      Boston, MA| ATL|   Atlanta, GA|        2015|
+----------+----------+-----------------+------+---------------+----+-------------+------------+
only showing top 5 rows

23/02/26 13:54:08 INFO spark_schema_demo: PARQUET Schema: struct<FL_DATE:date,OP_CARRIER:string,OP_
```

INFO spark_schema_demo: PARQUET Schema:
struct<FL_DATE:**date**,OP_CARRIER:string,OP_CARRIER_FL_NUM:int,ORIGIN:string,ORIGIN_CITY_NAME:string,DEST:string,DEST_CITY_NAME:string,CRS_DEP_TIME:int,DEP_TIME:int,WHEELS_ON:int,TAXI_IN:int,CRS_ARR_TIME:int,ARR_TIME:int,CANCELLED:int,DISTANCE:int>

- **Parquet file format is recommended for spark.**

- **Creating Spark DataFrame and Dataset Transformations**

  - As we see in previous example the schema is not working with csv and json perfectly.
  - Now we are **explicitly** setting schema for DF with spark data types.

Spark Data Types : https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.types

S.No.	Spark Types	Scala Types	Python Types
1.	IntegerType	Int	Int
2.	LongType	Long	long
3.	FloatType	Float	Float
4.	DoubleType	Double	Float
5.	StringType	String	String
6.	DateType	java.sql.Date	datetime.date
8.	TimestampType	java.sql.Timestamp	datetime.datetime
9.	ArrayType	scala.collection.Seq	List, tuple, or array
10.	MapType	scala.collection.Map	dict

  - Spark is work as compiler it complies with the high-level API code into low level RDD operations. During this compilation process it generate different execution plans and also perform bunch of optimizations it's not possible to spark engine to without maintaining its type information this kind of thing we seen in SQL every column contains a data type.

How to define Schema?

Spark Schema

1. Programmatically
2. Using DDL String

1. main.py Programmatically (used for CSV also used for other formats)

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StringType, IntegerType, DateType, StructType

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .master("local[3]") \
        .appName("spark_schema_demo") \
        .getOrCreate()

    logger = Log4j(spark)

    flightSchemaStruct = StructType([  # StructType:: DF row structure
        StructField("FL_DATE", DateType()),  # StructField:: Column definition
        StructField("OP_CARRIER", StringType()),
        StructField("OP_CARRIER_FL_NUM", IntegerType()),
        StructField("ORIGIN", StringType()),
        StructField("ORIGIN_CITY_NAME", StringType()),
        StructField("DEST", StringType()),
        StructField("DEST_CITY_NAME", StringType()),
        StructField("CRS_DEP_TIME", IntegerType()),
        StructField("DEP_TIME", IntegerType()),
        StructField("WHEELS_ON", IntegerType()),
        StructField("TAXI_IN", IntegerType()),
        StructField("CRS_ARR_TIME", IntegerType()),
        StructField("ARR_TIME", IntegerType()),
        StructField("CANCELLED", IntegerType()),
        StructField("DISTANCE", IntegerType())
    ])

    flight_time_csv_df = spark.read \
        .format("csv")\
        .option("header", "true")\
        .schema(flightSchemaStruct)\
        .option("mode", "FAILFAST")\
        .option("dateFormat", "M/d/y")\
        .load("data/flight*.csv")
     # .option("inferSchema", "true")\  here we provided data type schema
     # .option("mode", "FAILFAST") is used to prevent error due to wrong data type we provided.
     # csv needs date format pattern

    flight_time_csv_df.show(5)
    logger.info("CSV Schema: " + flight_time_csv_df.schema.simpleString())

    flight_time_json_df = spark.read \
        .format("json") \
        .load("data/flight*.json")

    flight_time_json_df.show(5)
    logger.info("JSON Schema: " + flight_time_json_df.schema.simpleString())

    flight_time_parquet_df = spark.read \
        .format("parquet") \
        .load("data/flight*.parquet")

    flight_time_parquet_df.show(5)
    logger.info("PARQUET Schema: " + flight_time_parquet_df.schema.simpleString())
```

Result

```
+----------+----------+----------------+------+---------------+----+----
|   FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|ORIGIN_CITY_NAME|DEST|DEST
+----------+----------+----------------+------+---------------+----+----
|2000-01-01|        DL|            1451|   BOS|      Boston, MA| ATL|   A
|2000-01-01|        DL|            1479|   BOS|      Boston, MA| ATL|   A
|2000-01-01|        DL|            1857|   BOS|      Boston, MA| ATL|   A
|2000-01-01|        DL|            1997|   BOS|      Boston, MA| ATL|   A
|2000-01-01|        DL|            2065|   BOS|      Boston, MA| ATL|   A
+----------+----------+----------------+------+---------------+----+----
only showing top 5 rows

23/02/26 14:23:39 INFO spark_schema_demo: CSV Schema: struct<FL_DATE:date,
```

After this data type schema we got correct data type for csv and JSON.

2. DDL String for Schema

```
flightSchemaDDL = """FL_DATE DATE, OP_CARRIER STRING, OP_CARRIER_FL_NUM INT, ORIGIN STRING,
      ORIGIN_CITY_NAME STRING, DEST STRING, DEST_CITY_NAME STRING, CRS_DEP_TIME INT, DEP_TIME INT,
      WHEELS_ON INT, TAXI_IN INT, CRS_ARR_TIME INT, ARR_TIME INT, CANCELLED INT, DISTANCE INT"""
```

Here we are using for JSON | Also used in other formats

```
from pyspark.sql import SparkSession

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .master("local[3]") \
        .appName("spark_schema_demo") \
        .getOrCreate()

    logger = Log4j(spark)

    flightSchemaDDL = """FL_DATE DATE, OP_CARRIER STRING, OP_CARRIER_FL_NUM INT, ORIGIN STRING,
          ORIGIN_CITY_NAME STRING, DEST STRING, DEST_CITY_NAME STRING, CRS_DEP_TIME INT, DEP_TIME INT,
          WHEELS_ON INT, TAXI_IN INT, CRS_ARR_TIME INT, ARR_TIME INT, CANCELLED INT, DISTANCE INT"""

    flight_time_json_df = spark.read \
        .format("json") \
        .schema(flightSchemaDDL)\
        .option("dateFormat", "M/d/y")\
        .load("data/flight*.json")

    flight_time_json_df.show(5)
    logger.info("JSON Schema: " + flight_time_json_df.schema.simpleString())
```

```
+----------+----------+----------------+------+---------------+----+-------------+--------
|   FL_DATE|OP_CARRIER|OP_CARRIER_FL_NUM|ORIGIN|ORIGIN_CITY_NAME|DEST|DEST_CITY_NAME|CRS_DEP_
+----------+----------+----------------+------+---------------+----+-------------+--------
|2000-01-01|        DL|            1451|   BOS|      Boston, MA| ATL|   Atlanta, GA|
|2000-01-01|        DL|            1479|   BOS|      Boston, MA| ATL|   Atlanta, GA|
|2000-01-01|        DL|            1857|   BOS|      Boston, MA| ATL|   Atlanta, GA|
|2000-01-01|        DL|            1997|   BOS|      Boston, MA| ATL|   Atlanta, GA|
|2000-01-01|        DL|            2065|   BOS|      Boston, MA| ATL|   Atlanta, GA|
+----------+----------+----------------+------+---------------+----+-------------+--------
only showing top 5 rows

23/02/26 14:35:51 INFO spark_schema_demo: JSON Schema: struct<FL_DATE:date,OP_CARRIER:string,
```

- **Spark DF Writer API**

DataFrameWriter API : https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameWriter

General structure

```
DataFrameWriter
    .format(...)
    .option(...)
    .partitionBy(...)
    .bucketBy(...)
    .sortBy(...)
    .save()
```

Indicative Example

```
DataFrame.write
         .format("parquet")
         .mode(saveMode)
         .option("path", "/data/flights/")
         .save()
```

Indicative Example

Built In Formats
CSV, JSON, Parquet, ORC, JDBC

```
DataFrame.write
         .format("parquet")
         .mode(saveMode)
         .option("path", "/data/flights/")
         .save()
```

parquet is default format

Indicative Example

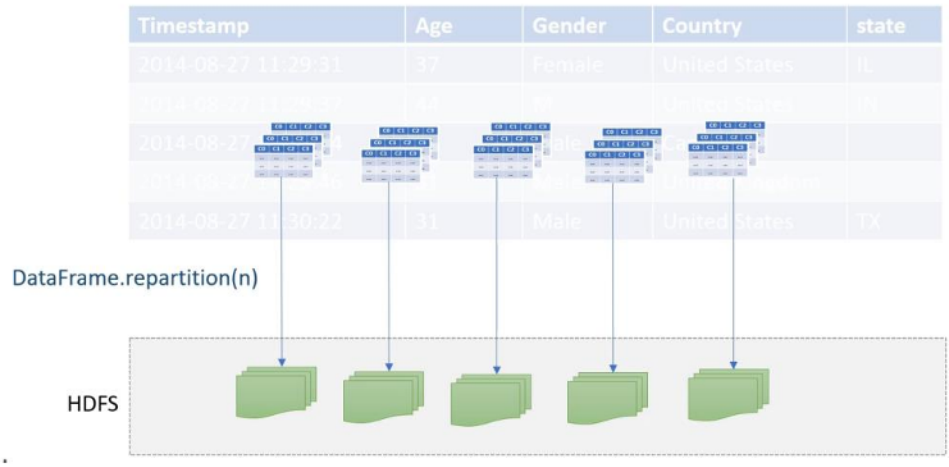
Read Mode
1. append
2. overwrite
3. errorIfExists
4. ignore

```
DataFrame.write
         .format("parquet")
         .mode(saveMode)
         .option("path", "/data/flights/")
         .save()
```

Saving modes

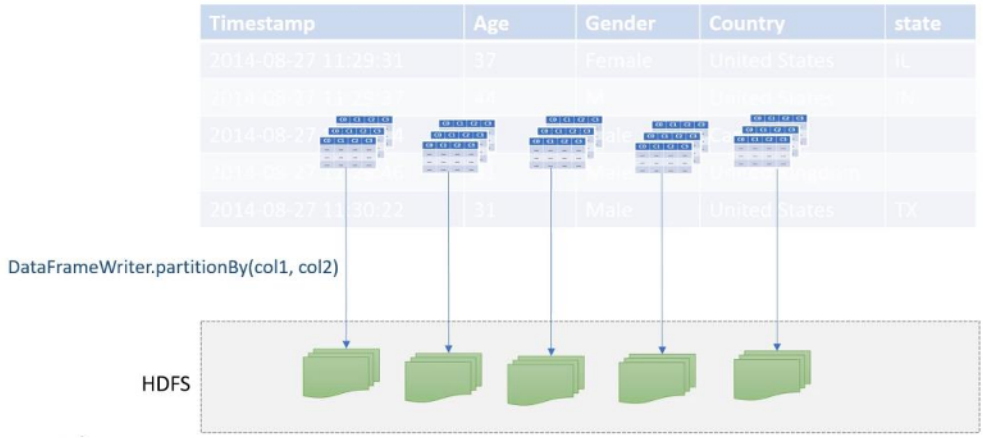
Spark File Layout

1. Number of files and file size
2. Partitions and Buckets
3. Sorted data



blind repartitioning

Partition by methods



key based partitioning

`DataFrameWriter.bucketBy(n, col1, col2)` **only available on spark managed tables**

- **Writing your Data and Managing Layout**

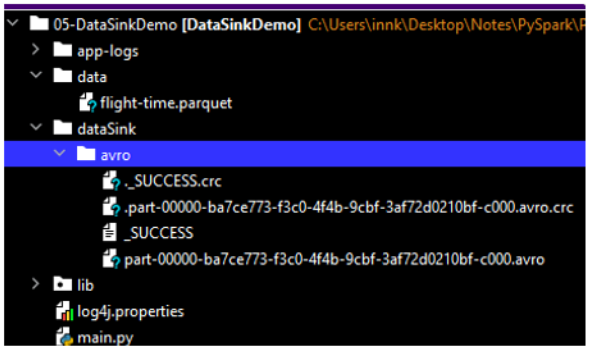
```
# spark.executor.extraJavaOptions  -XX:+PrintGCDetails -Dkey=value -Dnumbers="one two
spark.driver.extraJavaOptions      -Dlog4j.configuration=file:log4j.properties -DSpar
spark.jars.packages                org.apache.spark:spark-avro_2.12:3.3.2
```

2.12 Scala version 3.3.2 is Pyspark version
https://spark.apache.org/docs/latest/sql-data-sources-avro.html#configuration

Adding some Scala packages in spart-defaults.conf by above line. this allows to add jar in dependencies. Now you can read and write AVRO files.

main.py (before default 2 partitions)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import spark_partition_id

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[3]")\
        .appName("SparkSchemaDemo")\
        .getOrCreate()

    logger = Log4j(spark)

    flightTimeParquetDF = spark.read\
        .format("parquet")\
        .load("data/flight-time.parquet")

    flightTimeParquetDF.write\
        .format("avro")\
        .mode("overwrite")\
        .option("path", "dataSink/avro/")\
        .save()

    logger.info("Num Partition before: " + str(flightTimeParquetDF.rdd.getNumPartitions())) # to count partitions
    flightTimeParquetDF.groupBy(spark_partition_id()).count().show() # After executing this code we got only one avro file but we expected 2 because we have 2 partitions by this count we will check that our all data is in 1st partition and 2nd don't have any data that's why we got only one ARVO file. To resolve this problem following modifications are done.
```




```
    0 artifacts copied, 3 already retrieved (0kB/29ms)
23/02/26 19:06:55 WARN ProcfsMetricsGetter: Exception when trying
23/02/26 19:06:57 INFO SparkSchemaDemo: Num Partition before: 2
```

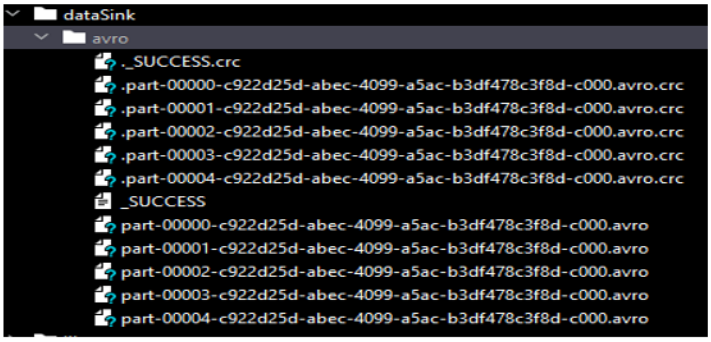
```
+-------------------+------+
|SPARK_PARTITION_ID()| count|
+-------------------+------+
|                  0|470477|
+-------------------+------+
```

main.py (After forced 5 partitions)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import spark_partition_id

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[3]")\
        .appName("SparkSchemaDemo")\
        .getOrCreate()

    logger = Log4j(spark)

    flightTimeParquetDF = spark.read\
        .format("parquet")\
        .load("data/flight-time.parquet")

    logger.info("Num Partition before: " + str(flightTimeParquetDF.rdd.getNumPartitions()))
    flightTimeParquetDF.groupBy(spark_partition_id()).count().show()

    partitionedDF = flightTimeParquetDF.repartition(5) # forced partitions
    logger.info("Num Partition after: " + str(partitionedDF.rdd.getNumPartitions()))
    partitionedDF.groupBy(spark_partition_id()).count().show()

    partitionedDF.write \
        .format("avro") \
        .mode("overwrite") \
        .option("path", "dataSink/avro/") \
        .save()
```

```
23/02/26 19:17:08 INFO SparkSchemaDemo: Num Partition after: 5
+-------------------+-----+
|SPARK_PARTITION_ID()|count|
+-------------------+-----+
|                  0|94096|
|                  1|94095|
|                  2|94095|
|                  3|94095|
|                  4|94096|
+-------------------+-----+
```

Repartition worked



we got 5 AVRO files

Some times partitioning not suitable but partitioning improves following.

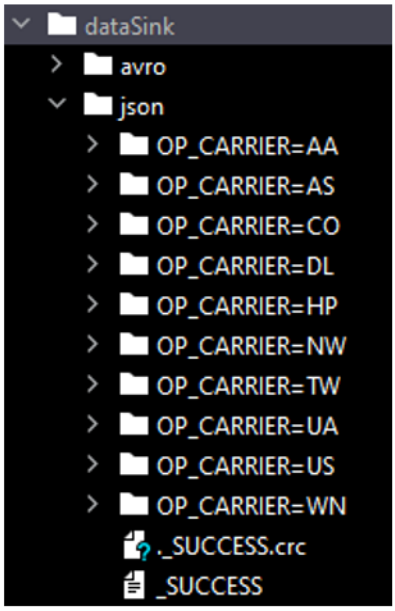
1. Parallel Processing
2. Partition Elimination

- **Partition BY method**

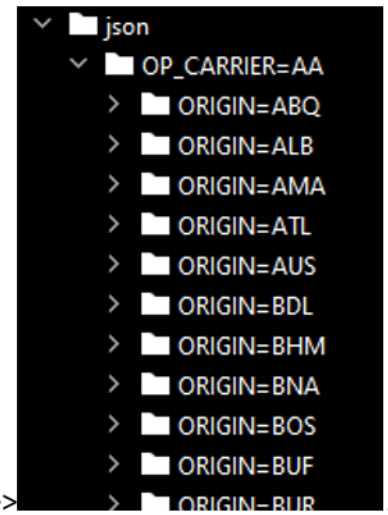
main.py

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import spark_partition_id

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[3]")\
        .appName("SparkSchemaDemo")\
        .getOrCreate()

    logger = Log4j(spark)

    flightTimeParquetDF = spark.read\
        .format("parquet")\
        .load("data/flight-time.parquet")

    flightTimeParquetDF.write\
        .format("json")\
        .mode("overwrite")\
        .option("path","dataSink/json/")\
        .partitionBy("OP_CARRIER", "ORIGIN")\
        .save()
```

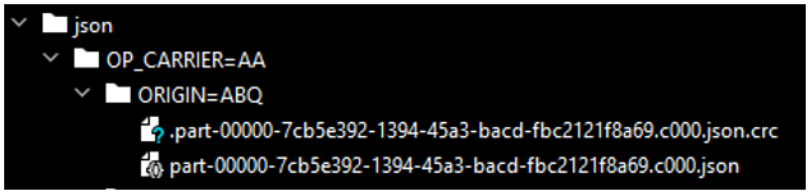
Here we got json format with partitioned by "OP_CARRIER", "ORIGIN" as follow



Origin>>



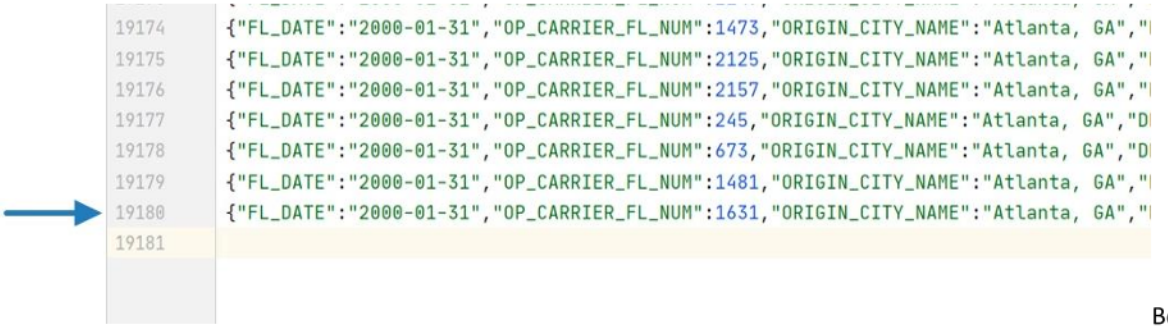
.....Sub Directory



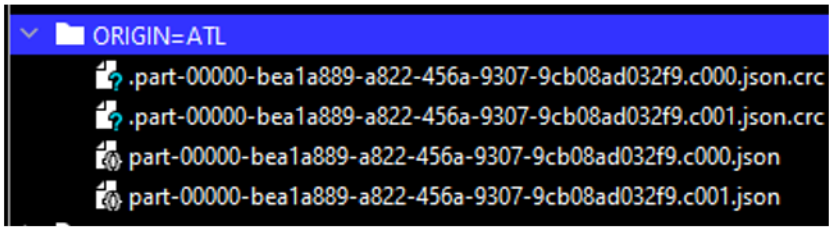
contains json

- **Handling file size (.option(“maxRecordsPerFile”, 10000))**

```
flightTimeParquetDF.write\
    .format("json")\
    .mode("overwrite")\
    .option("path","dataSink/json/")\
    .partitionBy("OP_CARRIER", "ORIGIN")\
    .option("maxRecordsPerFile", 10000)\
    .save()
```



Before



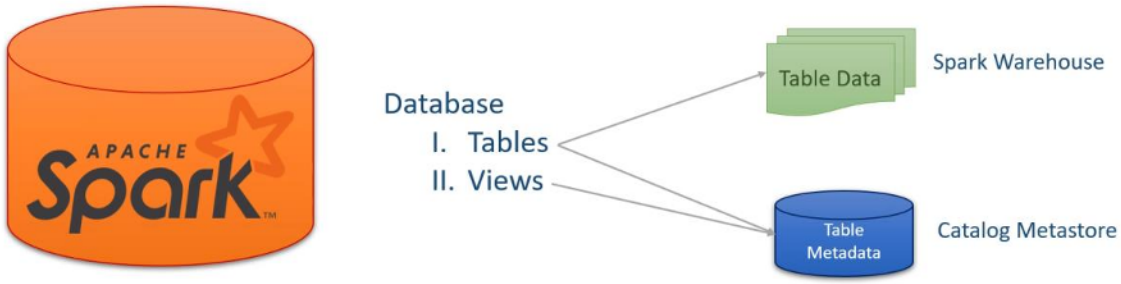
After two json files

1000 limit

What we learn

1. How to use DataFrameWriter API
2. How to use PartitionBy(...)
3. How to control file size using maxRecordsPerFile

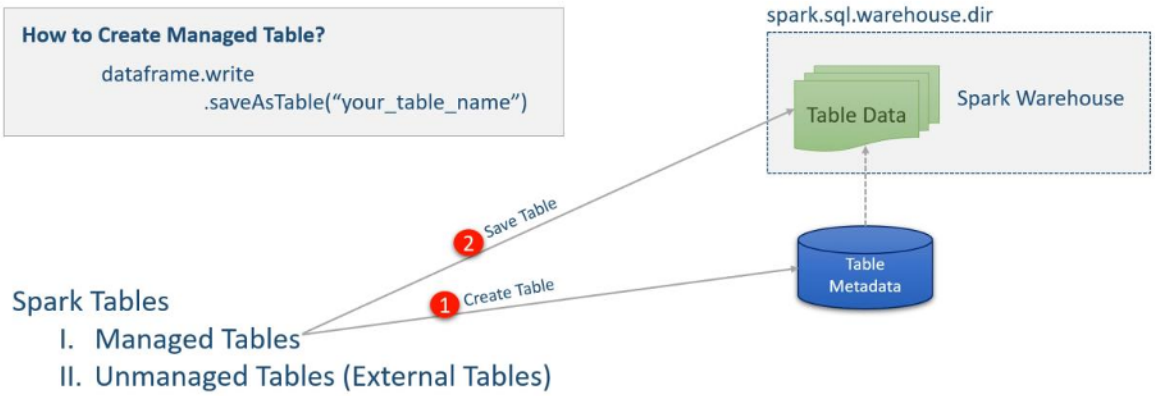
- **Spark Databases and Tables**
  - Apache Spark is database itself you can create database in it.



-

How to Create Managed Table?

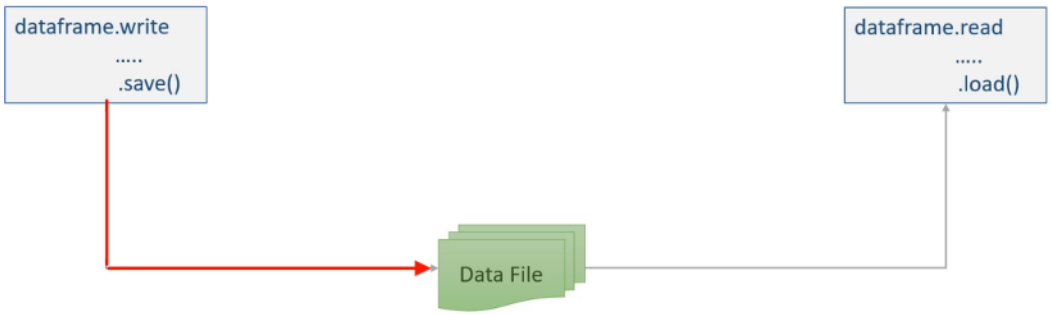
```
dataframe.write
            .saveAsTable("your_table_name")
```



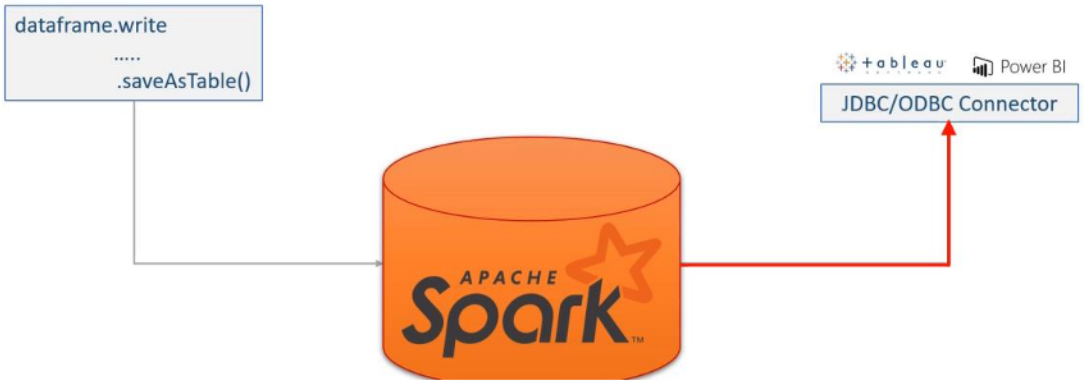
How to Create Unmanaged Table?

```
CREATE TABLE your_tbl_name (col1  data_type,
                            col2  data_type,
                            ……)
USING PARQUET
LOCATION "data_file_location"
```

- **Working with Spark SQL Table**
  - Need Hieve for this example.
  - Create manage table and store DF into spark table.
  - In real time scenario you will process your data and save the output data frame.



parquet or avro format

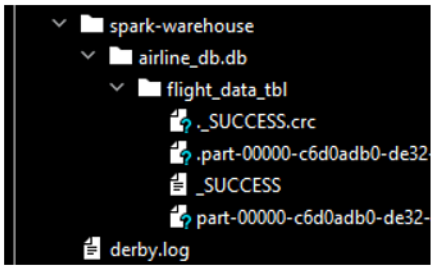


main.py

```
from pyspark.sql import *

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[3]")\
        .appName("SparkSQLTableDemo")\
        .enableHiveSupport()\
        .getOrCreate()

    logger = Log4j(spark)

    flightTimeParquetDF = spark.read\
        .format("parquet")\
        .load("dataSource/")

    spark.sql("CREATE DATABASE IF NOT EXISTS AIRLINE_DB")
    spark.catalog.setCurrentDatabase("AIRLINE_DB")

    # Create manage table and store DF into spark table(without any processing)

    flightTimeParquetDF.write\
        .mode("overwrite")\
        .saveAsTable("flight_data_tbl")

    logger.info(spark.catalog.listTables("AIRLINE_DB"))
```

```
23/02/27 14:16:59 INFO SparkSQLTableDemo: [[flight_data_tbl, airline_db, null, MANAGED, false]]

Process finished with exit code 0
```

Here is catalog information of DB



Partition by method

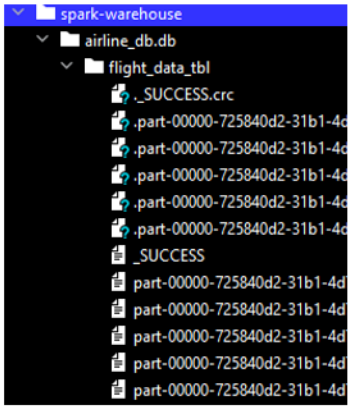
```
flightTimeParquetDF.write\
    .mode("overwrite")\
    .partitionBy("ORIGIN", "OP_CARRIER")\
    .saveAsTable("flight_data_tbl")
```

here we got 200+ dir.

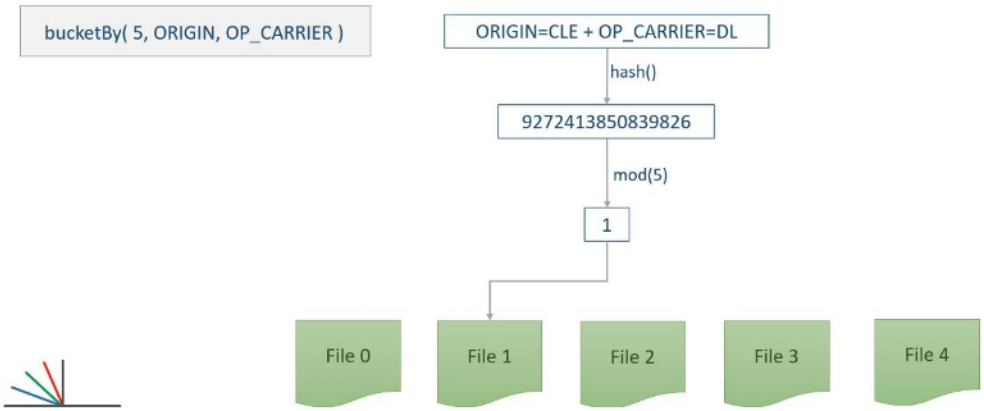
So, When Table contains to many unique values you need to use bucket by

- The bucket by allows you to restrict the partitions.

```
flightTimeParquetDF.write\
    .format("csv")\ # here parquet is recommended format we are using csv only for investing data
    .mode("overwrite")\
    .bucketBy(5, "ORIGIN", "OP_CARRIER")\
    .saveAsTable("flight_data_tbl")
```



Here we got 5 CSV files (bucket)



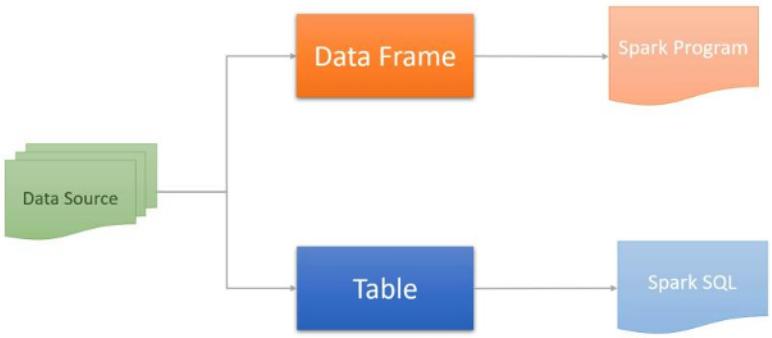
Sort by

```
flightTimeParquetDF.write\
    .format("csv")\
    .mode("overwrite")\
    .bucketBy(5, "ORIGIN", "OP_CARRIER")\
    .sortBy("ORIGIN", "OP_CARRIER")\
    .saveAsTable("flight_data_tbl")
```

here we got sorted values columns in bucket

```
62  2000-01-10,AA,1182,ABQ,"Albuquerque, N
63  2000-01-10,AA,1216,ABQ,"Albuquerque, N
64  2000-01-10,AA,1306,ABQ,"Albuquerque, N
65  2000-01-10,AA,1400,ABQ,"Albuquerque, N
66  2000-01-10,AA,1498,ABQ,"Albuquerque, N
67  2000-01-11,AA,438,ABQ,"Albuquerque, NM
68  2000-01-11,AA,1182,ABQ,"Albuquerque, N
69  2000-01-11,AA,1216,ABQ,"Albuquerque, N
```

- **Spark Dataframe and Dataset Transformation** ref.48



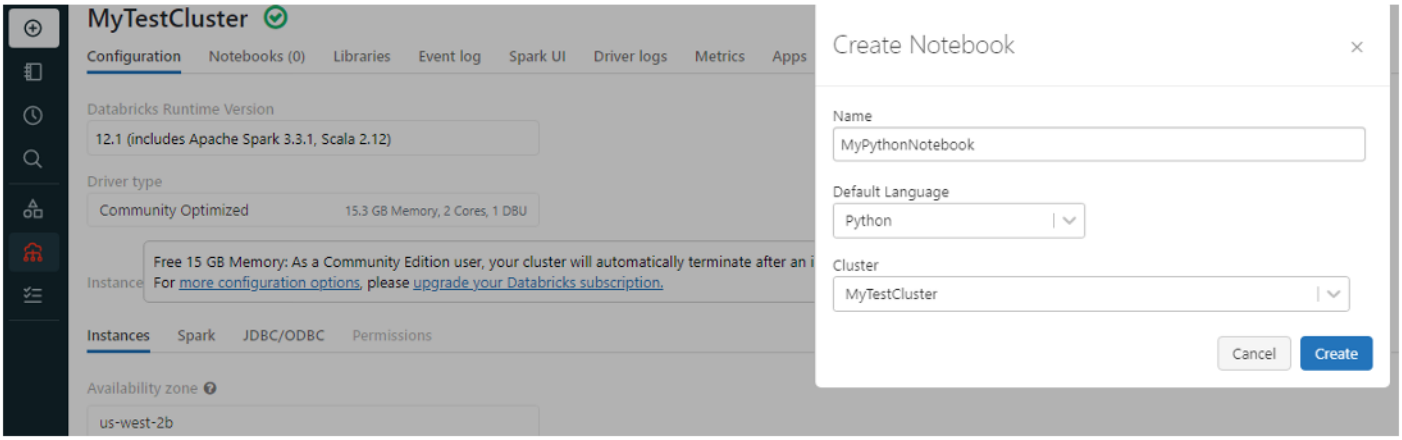
Dataframe = Dataset[Row]

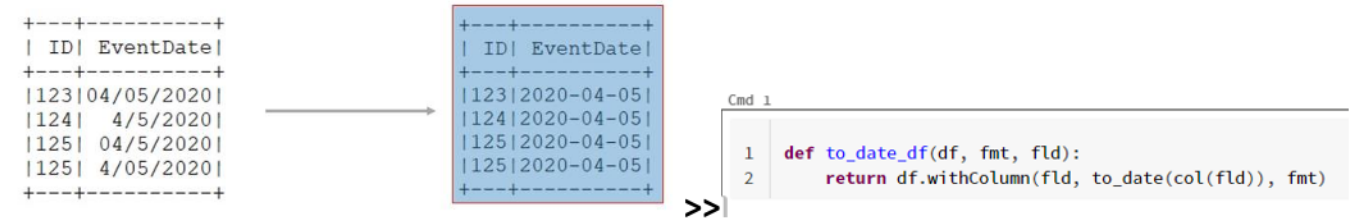
Timestamp	Age	Gender	Country	state
2014-08-27 11:29:31	37	Female	United States	IL
2014-08-27 11:29:37	44	M	United States	IN
2014-08-27 11:29:44	32	Male	Canada	
2014-08-27 11:29:46	31	Male	United Kingdom	
2014-08-27 11:30:22	31	Male	United States	TX

1. Manually creating Rows and Dataframe.
2. Collecting Dataframe rows to the driver.
3. Work with an individual row in Spark Transformations.

- **Working with DataFrame Rows**

Manually creating Rows and Dataframe





1. A large project might need hundreds of small sample data files to test your functions.
2. Your build pipeline might run slow due to loading hundreds of sample files and increased I/O.

Databricks code:

```
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark.sql.types import *

def to_date_df(df, fmt, fld):
  return df.withColumn(fld, to_date(col(fld), fmt))

my_schema = StructType([
  StructField("ID", StringType()),
  StructField("EventDate", StringType())])

my_rows = [Row("123", "04/05/2020"), Row("124", "4/5/2020"), Row("125", "04/5/2020"), Row("126", "4/05/2020")]
my_rdd = spark.sparkContext.parallelize(my_rows, 2)
my_df = spark.createDataFrame(my_rdd, my_schema)

my_df.printSchema()
my_df.show()
```

```
root
 |-- ID: string (nullable = true)
 |-- EventDate: string (nullable = true)

+---+----------+
| ID| EventDate|
+---+----------+
|123|04/05/2020|
|124| 4/5/2020|
|125| 04/5/2020|
|126| 4/05/2020|
+---+----------+
```

```
new_df = to_date_df(my_df, "M/d/y", "EventDate")
new_df.printSchema()
new_df.show()
```

```
root
 |-- ID: string (nullable = true)
 |-- EventDate: date (nullable = true)

+---+----------+
| ID| EventDate|
+---+----------+
|123|2020-04-05|
|124|2020-04-05|
|125|2020-04-05|
|126|2020-04-05|
+---+----------+
```

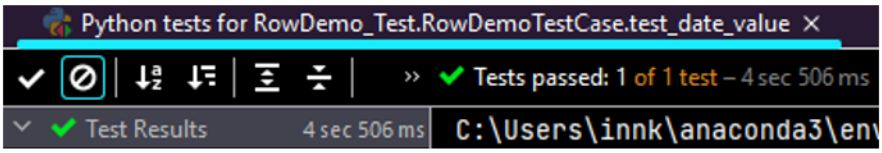
Collecting Dataframe rows to the driver.

- **DataFrame Rows and Unit testing**

ref. **RowDemo** for **main.py** in PyCharm with above same code

Unit Test for above code

```
from datetime import date
from unittest import TestCase

from pyspark.sql import *
from pyspark.sql.types import *

from RowDemo import to_date_df


class RowDemoTestCase(TestCase):

    @classmethod
    def setUpClass(cls) -> None:
        cls.spark = SparkSession.builder \
            .master("local[3]") \
            .appName("RowDemoTest") \
            .getOrCreate()

        my_schema = StructType([
            StructField("ID", StringType()),
            StructField("EventDate", StringType())])

        my_rows = [Row("123", "04/05/2020"), Row("124", "4/5/2020"), Row("125", "04/5/2020"), Row("126", "4/05/2020")]
        my_rdd = cls.spark.sparkContext.parallelize(my_rows, 2)
        cls.my_df = cls.spark.createDataFrame(my_rdd, my_schema)

    def test_data_type(self):
        rows = to_date_df(self.my_df, "M/d/y", "EventDate").collect()
        for row in rows:
            self.assertIsInstance(row["EventDate"], date)

    def test_date_value(self):
        rows = to_date_df(self.my_df, "M/d/y", "EventDate").collect()
        for row in rows:
            self.assertEqual(row["EventDate"], date(2020, 4, 5))
```



Work with an individual row in Spark Transformations.

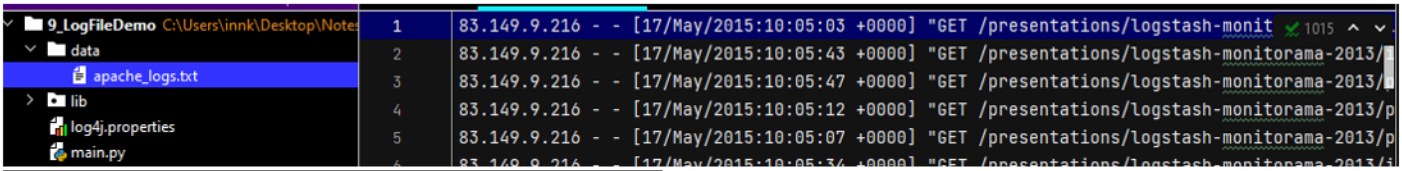
- **DataFrame Rows and Unstructured Data**

  - **Spark DF offers bunch of Transformation functions.**

```
agg(*exprs)
cov(col1, col2)
crosstab(col1, col2)
cube(*cols)
filter(condition)
groupBy(*cols)
join(other, on=None, how=None)
orderBy(*cols, **kwargs)
replace(to_replace, value, subset)
rollup(*cols)
select(*cols)
sort(*cols, **kwargs)
where(condition)
withColumn(colName, col)
avg(*cols)
max(*cols)
mean(*cols)
min(*cols)
sum(*cols)
```

  - When your DF doesn't have column structure Then you have to create columnar structure and then use Transformation functions.

Here we have Apache Log Server File(which contain some pattern and it unstructured):




```
file_df = spark.read.text("data/apache_logs.txt")
file_df.printSchema()
```

```
|-- value: string (nullable = true)
```

we have only one string field "value".

log_reg = r'^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+) (\S+)" (\d{3}) (\S+) "(\S+)" "([^"]*)'
this above regx gives following filed

1. IP
2. client.
3. user
4. datetime
5. cmd
6. request
7. protocol
8. status
9. bytes
10. referrer
11. userAgent

>>

```
logs_df = file_df.select(regexp_extract('value', log_reg, 1).alias('ip'),
                         regexp_extract('value', log_reg, 4).alias('date'),
                         regexp_extract('value', log_reg, 6).alias('request'),
                         regexp_extract('value', log_reg, 10).alias('referrer'))

logs_df.printSchema()
```

>>>

```
|-- ip: string (nullable = true)
|-- date: string (nullable = true)
|-- request: string (nullable = true)
|-- referrer: string (nullable = true)
```

Now we got 4 schema

```
logs_df.printSchema()

logs_df.groupBy("referrer")\
    .count()\
    .show(100, truncate=False)
```

```
+-----------------------------------------------
|referrer
+-----------------------------------------------
|http://www.semicomplete.com/blog/tags/wifi
|http://manpages.ubuntu.com/manpages/lucid/man1/
|http://s-chassis.co.nz/viewtopic.php?f=16&t=926
|https://www.google.sk/
|http://www.semicomplete.com/projects/keynav/
|http://superuser.com/questions/355151/is-it-pos
|https://www.google.ca/
```

but we got complete urls so following

main.py

```
from pyspark.sql import *
from pyspark.sql.functions import regexp_extract, substring_index

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .master("local[3]") \
        .appName("LogFileDemo") \
        .getOrCreate()

    file_df = spark.read.text("data/apache_logs.txt")

    log_reg = r'^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+) (\S+)" (\d{3}) (\S+) "(\S+)" "([^"]*)'

    logs_df = file_df.select(regexp_extract('value', log_reg, 1).alias('ip'),
                             regexp_extract('value', log_reg, 4).alias('date'),
                             regexp_extract('value', log_reg, 6).alias('request'),
                             regexp_extract('value', log_reg, 10).alias('referrer'))

    logs_df.printSchema()

    logs_df \
        .where("trim(referrer) != '-' ") \
        .withColumn("referrer", substring_index("referrer", "/", 3)) \
        .groupBy("referrer") \
        .count() \
        .show(100, truncate=False)
```

```
+-----------------------------------+-----+
|referrer                           |count|
+-----------------------------------+-----+
|http://ijavascript.cn              |1    |
|http://www.google.co.tz            |1    |
|http://www.google.ca               |6    |
|https://www.google.hr              |2    |
|https://www.google.ch              |1    |
|http://www.google.ru               |6    |
|http://www.raspberrypi-spanish.es  |1    |
|http://semicomplete.com            |2001 |
|http://manpages.ubuntu.com         |2    |
|http://www.google.fi               |4    |
|-                                  |4073 |
|https://www.google.co.za           |1    |
```

.where("trim(referrer) != '-' ") \

- **Working with DataFrame Columns**

Column

Timestamp	Age	Gender	Country	state
2014-08-27 11:29:31	37	Female	United States	IL
2014-08-27 11:29:37	44	M	United States	IN
2014-08-27 11:29:44	32	Male	Canada	
2014-08-27 11:29:46	31	Male	United Kingdom	
2014-08-27 11:30:22	31	Male	United States	TX

1. What is a Column and How to reference it?
2. How to create column expressions?

DATA BRICKS

Cmd 1

```
%fs ls /databricks-datasets/
```

Table +

	path
1	dbfs:/databricks-datasets/
2	dbfs:/databricks-datasets/COVID/
3	dbfs:/databricks-datasets/README.md
4	dbfs:/databricks-datasets/Rdatasets/
5	dbfs:/databricks-datasets/SPARK_README.md
6	dbfs:/databricks-datasets/adult/
7	dbfs:/databricks-datasets/airlines/

56 rows | 51.75 seconds runtime

Command took 51.75 seconds -- by innksn@gmail.com at 2/27/2023, 9

we can get this dataset which are provided by databricks.

Cmd 2

```
%fs ls /databricks-datasets/airlines/
```

Table +

	path	nam
1	dbfs:/databricks-datasets/airlines/README.md	REA
2	dbfs:/databricks-datasets/airlines/_SUCCESS	_SU
3	dbfs:/databricks-datasets/airlines/part-00000	part
4	dbfs:/databricks-datasets/airlines/part-00001	part
5	dbfs:/databricks-datasets/airlines/part-00002	part
6	dbfs:/databricks-datasets/airlines/part-00003	part
7	dbfs:/databricks-datasets/airlines/part-00004	part

1,000 rows | Truncated data | 2.86 seconds runti

Command took 2.86 seconds -- by innksn@gmail.com at 2/27/20

Cmd 3

```
%fs head /databricks-datasets/airlines/part-00000
```

```
[Truncated to first 65536 bytes]
Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTime,UniqueCarrier,FlightNum,TailNum,Actual
rted,CarrierDelay,WeatherDelay,NASDelay,SecurityDelay,LateAircraftDelay,IsArrDelayed,IsDepDelayed
1987,10,14,3,741,730,912,849,PS,1451,NA,91,79,NA,23,11,SAN,SFO,447,NA,NA,0,NA,0,NA,NA,NA,NA,NA,YES,YES
1987,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,-1,SAN,SFO,447,NA,NA,0,NA,0,NA,NA,NA,NA,NA,YES,NO
1987,10,17,6,741,730,918,849,PS,1451,NA,97,79,NA,29,11,SAN,SFO,447,NA,NA,0,NA,0,NA,NA,NA,NA,NA,YES,YES
1987,10,18,7,729,730,847,849,PS,1451,NA,78,79,NA,-2,-1,SAN,SFO,447,NA,NA,0,NA,0,NA,NA,NA,NA,NA,NO,NO
1987,10,19,1,749,730,922,849,PS,1451,NA,93,79,NA,33,19,SAN,SFO,447,NA,NA,0,NA,0,NA,NA,NA,NA,NA,YES,YES
```

Cmd 4

```
airlinesDF = spark.read \
.format("csv") \
.option("header", "true") \
.option("inferSchema","true") \
.option("samplingRatio", "0.0001") \
.load("/databricks-datasets/airlines/part-00000")
```

Cmd 5

```
airlinesDF.select("Origin", "Dest", "Distance" ).show(10)
```

▸ (1) Spark Jobs

```
+------+----+--------+
|Origin|Dest|Distance|
+------+----+--------+
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
+------+----+--------+
only showing top 10 rows
```

Column String

Cmd 6

```
from pyspark.sql.functions import *
airlinesDF.select(column("Origin"), col("Dest"), "Distance").show(10)
```

▸ (1) Spark Jobs

```
+------+----+--------+
|Origin|Dest|Distance|
+------+----+--------+
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
|   SAN| SFO|     447|
+------+----+--------+
only showing top 10 rows
```

Column Object

Cmd 7

```
airlinesDF.select("Origin", "Dest", "Distance", "Year","Month","DayofMonth").show(10)
```

▸ (1) Spark Jobs

```
+------+----+--------+----+-----+----------+
|Origin|Dest|Distance|Year|Month|DayofMonth|
+------+----+--------+----+-----+----------+
|   SAN| SFO|     447|1987|   10|        14|
|   SAN| SFO|     447|1987|   10|        15|
|   SAN| SFO|     447|1987|   10|        17|
|   SAN| SFO|     447|1987|   10|        18|
|   SAN| SFO|     447|1987|   10|        19|
|   SAN| SFO|     447|1987|   10|        21|
|   SAN| SFO|     447|1987|   10|        22|
|   SAN| SFO|     447|1987|   10|        23|
|   SAN| SFO|     447|1987|   10|        24|
|   SAN| SFO|     447|1987|   10|        25|
+------+----+--------+----+-----+----------+
only showing top 10 rows
```

by using column object

→ 1. String Expressions or SQL Expressions
2. Column Object Expressions

Cmd 8

```
airlinesDF.select("Origin", "Dest", "Distance", expr("to_date(concat(Year,Month,DayofMonth),'yyyyMMdd') as FlightDate")).show(10)
```

▸ (1) Spark Jobs

```
+------+----+--------+----------+
|Origin|Dest|Distance|FlightDate|
+------+----+--------+----------+
|   SAN| SFO|     447|1987-10-14|
|   SAN| SFO|     447|1987-10-15|
|   SAN| SFO|     447|1987-10-17|
|   SAN| SFO|     447|1987-10-18|
|   SAN| SFO|     447|1987-10-19|
|   SAN| SFO|     447|1987-10-21|
|   SAN| SFO|     447|1987-10-22|
|   SAN| SFO|     447|1987-10-23|
|   SAN| SFO|     447|1987-10-24|
|   SAN| SFO|     447|1987-10-25|
+------+----+--------+----------+
only showing top 10 rows
```

Cmd 8

```
airlinesDF.selectExpr("Origin", "Dest", "Distance", "to_date(concat(Year,Month,DayofMonth),'yyyyMMdd') as FlightDate").show(10)
```

▸ (1) Spark Jobs

```
+------+----+--------+----------+
|Origin|Dest|Distance|FlightDate|
+------+----+--------+----------+
|   SAN| SFO|     447|1987-10-14|
|   SAN| SFO|     447|1987-10-15|
|   SAN| SFO|     447|1987-10-17|
|   SAN| SFO|     447|1987-10-18|
|   SAN| SFO|     447|1987-10-19|
|   SAN| SFO|     447|1987-10-21|
|   SAN| SFO|     447|1987-10-22|
|   SAN| SFO|     447|1987-10-23|
|   SAN| SFO|     447|1987-10-24|
|   SAN| SFO|     447|1987-10-25|
+------+----+--------+----------+
only showing top 10 rows
```

By using Column Object expression ref.52 / 10 folder

Cmd 9

```
airlinesDF.select("Origin", "Dest", "Distance", to_date(concat("Year","Month","DayofMonth"),"yyyyMMdd").alias("FlightDate")).show(10)
```

▸ (1) Spark Jobs

```
+------+----+--------+----------+
|Origin|Dest|Distance|FlightDate|
+------+----+--------+----------+
|   SAN| SFO|     447|1987-10-14|
|   SAN| SFO|     447|1987-10-15|
|   SAN| SFO|     447|1987-10-17|
|   SAN| SFO|     447|1987-10-18|
|   SAN| SFO|     447|1987-10-19|
|   SAN| SFO|     447|1987-10-21|
|   SAN| SFO|     447|1987-10-22|
|   SAN| SFO|     447|1987-10-23|
|   SAN| SFO|     447|1987-10-24|
|   SAN| SFO|     447|1987-10-25|
+------+----+--------+----------+
only showing top 10 rows
```

- **Creating and Using UDF(User Defined Functions)**

To show DF

```
from pyspark.sql import SparkSession

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("UDF Demo") \
        .master("local[2]") \
        .getOrCreate()

    logger = Log4j(spark)

    survey_df = spark.read \
        .option("header", "true") \
        .option("inferSchema", "true") \
        .csv("data/survey.csv")

    survey_df.show(10)
```

```
|          Timestamp|Age|Gender|       Country|stat
+-------------------+---+------+--------------+----
|2014-08-27 11:29:31| 37|Female| United States|   I
|2014-08-27 11:29:37| 44|     M| United States|   I
|2014-08-27 11:29:44| 32|  Male|        Canada|   N
|2014-08-27 11:29:46| 31|  Male|United Kingdom|   N
|2014-08-27 11:30:22| 31|  Male| United States|   T
|2014-08-27 11:31:22| 33|  Male| United States|   T
|2014-08-27 11:31:50| 35|Female| United States|   M
|2014-08-27 11:32:05| 39|     M|        Canada|   N
|2014-08-27 11:32:39| 42|Female| United States|   I
|2014-08-27 11:32:43| 23|  Male|        Canada|   N
+-------------------+---+------+--------------+----
only showing top 10 rows
```

```
import re

from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, expr
from pyspark.sql.types import StringType

from lib.logger import Log4j


def parse_gender(gender): #UDF
    female_pattern = r"^f$|f.m|w.m" # female ^f$ or f.m or w.m women
    male_pattern = r"^m$|ma|m.l" # male ^m$ or ma or m.l
    if re.search(female_pattern, gender.lower()):
        return "Female"
    elif re.search(male_pattern, gender.lower()):
        return "Male"
    else:
        return "Unknown"

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("UDF Demo") \
        .master("local[2]") \
        .getOrCreate()

    logger = Log4j(spark)

    survey_df = spark.read \
        .option("header", "true") \
        .option("inferSchema", "true") \
        .csv("data/survey.csv")

    survey_df.show(10)

    #Column Object Expression
    # registering to column object expression.
    parse_gender_udf = udf(parse_gender, returnType=StringType())
    logger.info("Catalog Entry:")
    [logger.info(r) for r in spark.catalog.listFunctions() if "parse_gender" in r.name]

    # withcolum is allows as to transform the column without impaction other column of DF.
    survey_df2 = survey_df.withColumn("Gender", parse_gender_udf("Gender"))
    survey_df2.show(10)

    #UDF reg. this registered as SQL function and also create one entry in catalog.
    spark.udf.register("parse_gender_udf", parse_gender, StringType())
    logger.info("Catalog Entry:")
    [logger.info(r) for r in spark.catalog.listFunctions() if "parse_gender" in r.name]

    survey_df3 = survey_df.withColumn("Gender", expr("parse_gender_udf(Gender)"))
    survey_df3.show(10)
```

```
+-------------------+---+------+
|          Timestamp|Age|Gender|
+-------------------+---+------+
|2014-08-27 11:29:31| 37|Female|
|2014-08-27 11:29:37| 44|     M|
|2014-08-27 11:29:44| 32|  Male|
|2014-08-27 11:29:46| 31|  Male|
|2014-08-27 11:30:22| 31|  Male|
|2014-08-27 11:31:22| 33|  Male|
|2014-08-27 11:31:50| 35|Female|
|2014-08-27 11:32:05| 39|     M|
|2014-08-27 11:32:39| 42|Female|
|2014-08-27 11:32:43| 23|  Male|
+-------------------+---+------+
only showing top 10 rows
```

Original DF O/p

```
+-------------------+---+------+
|          Timestamp|Age|Gender|
+-------------------+---+------+
|2014-08-27 11:29:31| 37|Female|
|2014-08-27 11:29:37| 44|  Male|
|2014-08-27 11:29:44| 32|  Male|
|2014-08-27 11:29:46| 31|  Male|
|2014-08-27 11:30:22| 31|  Male|
|2014-08-27 11:31:22| 33|  Male|
|2014-08-27 11:31:50| 35|Female|
|2014-08-27 11:32:05| 39|  Male|
|2014-08-27 11:32:39| 42|Female|
|2014-08-27 11:32:43| 23|  Male|
+-------------------+---+------+
only showing top 10 rows
```

Transformed DF with Column Object

```
+-------------------+---+------+
|          Timestamp|Age|Gender|
+-------------------+---+------+
|2014-08-27 11:29:31| 37|Female|
|2014-08-27 11:29:37| 44|  Male|
|2014-08-27 11:29:44| 32|  Male|
|2014-08-27 11:29:46| 31|  Male|
|2014-08-27 11:30:22| 31|  Male|
|2014-08-27 11:31:22| 33|  Male|
|2014-08-27 11:31:50| 35|Female|
|2014-08-27 11:32:05| 39|  Male|
|2014-08-27 11:32:39| 42|Female|
|2014-08-27 11:32:43| 23|  Male|
+-------------------+---+------+
only showing top 10 rows
```

Transformed DF with SQL Expression

- **Misc(Miscellaneous) Transformation**

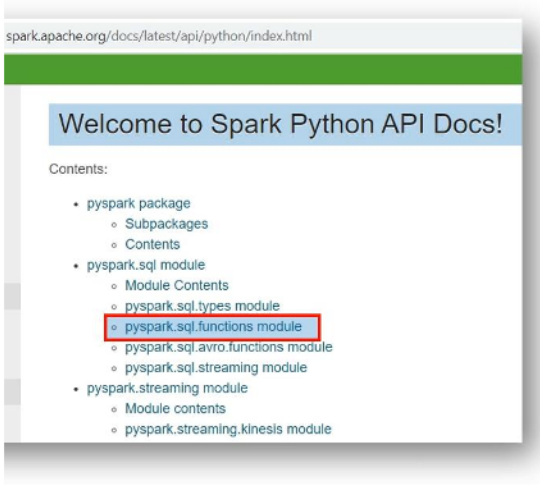
Miscellaneous

1. Quick method to create Dataframes
2. Adding monotonically increasing id
3. Using Case When Then transformation
4. Casting your columns
5. Adding columns to Dataframes
6. Dropping Columns
7. Dropping duplicate rows
8. Sorting Dataframes

ref. 54

- **Aggregation in Apache Spark**
  1. Simple Aggregations
  2. Grouping Aggregations
  3. Windowing Aggregations

this are built works via built in function.



Aggregating Functions

avg()
count()
max()
min()
sum()

Window Functions

lead()
lag()
rank()
dense_rank()
cume_dist()

main.py

```
from pyspark.sql import SparkSession, functions as f

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[2]")\
        .appName("AggFun")\
        .getOrCreate()

    logger = Log4j(spark)

    invoice_df = spark.read\
        .format("csv")\
        .option("header", "true")\
        .option("inferSchema", "true")\
        .load("data/invoices.csv")

    invoice_df.select(f.count("*").alias("Count *"),
                      f.sum("Quantity").alias("TotalQuantity"),
                      f.avg("UnitPrice").alias("AvgPrice"),
                      f.countDistinct("InvoiceNo").alias("CountDistinct")).show()
```

By Object Function

```
+-------+-------------+-----------------+-------------+
|Count *|TotalQuantity|         AvgPrice|CountDistinct|
+-------+-------------+-----------------+-------------+
| 541909|      5176450|4.611113626088498|        25900|
+-------+-------------+-----------------+-------------+
```

Simple Aggregation

main.py

```
from pyspark.sql import SparkSession, functions as f

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[2]")\
        .appName("AggFun")\
        .getOrCreate()

    logger = Log4j(spark)

    invoice_df = spark.read\
        .format("csv")\
        .option("header", "true")\
        .option("inferSchema", "true")\
        .load("data/invoices.csv")

    invoice_df.selectExpr(
        "count(1) as `count 1`",
        "count(StockCode) as `count field`",
        "sum(Quantity) as TotalQuantity",
        "avg(UnitPrice) as AvgPrice"
    ).show()
```

By using SQL Expression

```
+-------+----------+-------------+---------------+
|count 1|count field|TotalQuantity|       AvgPrice|
+-------+----------+-------------+---------------+
| 541909|    541908|      5176450|4.61111362608295|
+-------+----------+-------------+---------------+
```

Count field is lesser than above because of in SQL count null are not countable.

Grouping Aggregations?

```
invoice_df.createOrReplaceTempView("sales")
    summary_sql = spark.sql("""
            SELECT Country, InvoiceNo,
                sum(Quantity) as TotalQuantity,
                round(sum(Quantity*UnitPrice),2) as InvoiceValue
            FROM sales
            GROUP BY Country, InvoiceNo""")

    summary_sql.show()
```

Group BY using Spark.SQL

```
+--------------+---------+-------------+------------+
|       Country|InvoiceNo|TotalQuantity|InvoiceValue|
+--------------+---------+-------------+------------+
|United Kingdom|   536446|          329|      440.89|
|United Kingdom|   536508|          216|      155.52|
|United Kingdom|   537018|           -3|         0.0|
|United Kingdom|   537401|          -24|         0.0|
|United Kingdom|   537811|           74|      268.86|
```

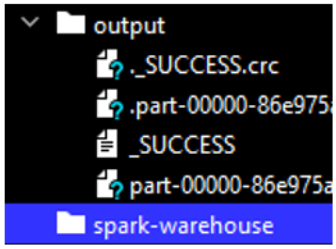
Above grouping by DF Expressions

```
summary_df = invoice_df \
      .groupBy("Country", "InvoiceNo") \
      .agg(f.sum("Quantity").alias("TotalQuantity"), #agg for aggregation
           f.round(f.sum(f.expr("Quantity * UnitPrice")), 2).alias("InvoiceValue"),
           f.expr("round(sum(Quantity * UnitPrice),2) as InvoiceValueExpr")
           )

   summary_df.show()
```

ref. 56 Grouping Aggregation***

```
from pyspark.sql import SparkSession, functions as f

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[2]")\
        .appName("AggFun")\
        .getOrCreate()

    logger = Log4j(spark)

    invoice_df = spark.read\
        .format("csv")\
        .option("header", "true")\
        .option("inferSchema", "true")\
        .load("data/invoices.csv")

    NumInvoices = f.countDistinct("InvoiceNo").alias("NumInvoices")
    TotalQuantity = f.sum("Quantity").alias("TotalQuantity")
    InvoiceValue = f.expr("round(sum(Quantity * UnitPrice),2) as InvoiceValue")

    exSummary_df = invoice_df \
        .withColumn("InvoiceDate", f.to_date(f.col("InvoiceDate"), "dd-MM-yyyy H.mm")) \
        .where("year(InvoiceDate) == 2010") \
        .withColumn("WeekNumber", f.weekofyear(f.col("InvoiceDate"))) \
        .groupBy("Country", "WeekNumber") \
        .agg(NumInvoices, TotalQuantity, InvoiceValue)

    exSummary_df.coalesce(1) \
        .write \
        .format("parquet") \
        .mode("overwrite") \
        .save("output")

    exSummary_df.sort("Country", "WeekNumber").show()
```

```
+--------------+----------+-----------+-------------+------------+
|       Country|WeekNumber|NumInvoices|TotalQuantity|InvoiceValue|
+--------------+----------+-----------+-------------+------------+
|     Australia|        48|          1|          107|      358.25|
|     Australia|        49|          1|          214|       258.9|
|     Australia|        50|          2|          133|      387.95|
|       Austria|        50|          2|            3|      257.04|
|       Bahrain|        51|          1|           54|      205.74|
|       Belgium|        48|          1|          528|       346.1|
|       Belgium|        50|          2|          285|      625.16|
```



- **Windowing Aggregations**

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
EIRE	48	7	2822	3147.23
EIRE	49	5	1280	3284.1
EIRE	50	5	1184	2321.78
EIRE	51	5	95	276.84
France	48	4	1299	2808.16
France	49	9	2303	4527.01
France	50	6	529	537.32
France	51	5	847	1702.87

>>

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
EIRE	48	7	2822	3147.23
EIRE	49	5	1280	3284.1
EIRE	50	5	1184	2321.78
EIRE	51	5	95	276.84

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue
France	48	4	1299	2808.16
France	49	9	2303	4527.01
France	50	6	529	537.32
France	51	5	847	1702.87

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	RunningTotal
EIRE	48	7	2822	3147.23	3147.23
EIRE	49	5	1280	3284.1	6431.33
EIRE	50	5	1184	2321.78	8753.11
EIRE	51	5	95	276.84	9029.95

Country	WeekNumber	NumInvoices	TotalQuantity	InvoiceValue	
France	48	4	1299	2808.16	2808.16
France	49	9	2303	4527.01	7335.17
France	50	6	529	537.32	7872.49
France	51	5	847	1702.87	9575.36

1. Identify your partitioning columns.
2. Identify your ordering requirement.
3. Define your window start and end.

main.py ref. 56

```
from pyspark.sql import SparkSession, Window
from pyspark.sql import functions as f

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("Agg Demo") \
        .master("local[2]") \
        .getOrCreate()

    logger = Log4j(spark)

    summary_df = spark.read.parquet("data/summary.parquet")

    running_total_window = Window.partitionBy("Country") \
        .orderBy("WeekNumber") \
        .rowsBetween(-2, Window.currentRow)

    summary_df.withColumn("RunningTotal",
                          f.sum("InvoiceValue").over(running_total_window)) \
        .show()
```

```
+---------+----------+-----------+-------------+-----------+------------------+
|  Country|WeekNumber|NumInvoices|TotalQuantity|InvoiceValue|      RunningTotal|
+---------+----------+-----------+-------------+-----------+------------------+
|Australia|        48|          1|          107|     358.25|            358.25|
|Australia|        49|          1|          214|      258.9|            617.15|
|Australia|        50|          2|          133|     387.95|1005.0999999999999|
|  Austria|        50|          2|            3|     257.04|            257.04|
|  Bahrain|        51|          1|           54|     205.74|            205.74|
|  Belgium|        48|          1|          528|      346.1|             346.1|
|  Belgium|        50|          2|          285|     625.16|            971.26|
|  Belgium|        51|          2|          942|     838.65|1809.9099999999999|
```

- **Spark Dataframe JOINS**

- **Dataframe Joins and Column name ambiguity**

```
orderDF
+--------+-------+----------+---+
|order_id|prod_id|unit_price|qty|
+--------+-------+----------+---+
|      01|     02|       350|  1|
|      01|     04|       580|  1|
|      01|     07|       320|  2|
|      02|     03|       450|  1|
|      02|     06|       220|  1|
|      03|     01|       195|  1|
|      04|     09|       270|  3|
|      04|     08|       410|  2|
|      05|     02|       350|  1|
+--------+-------+----------+---+
```

```
productDF
+-------+--------------------+----------+-------+
|prod_id|                name|list_price| qty   |
+-------+--------------------+----------+-------+
|     01|       Scroll Mouse|       250| 20    |
|     02|      Optical Mouse|       350| 20    |
|     03|     Wireless Mouse|       450| 50    |
|     04| Wireless Keyboard|       580| 50    |
|     05| Standard Keyboard|       360| 10    |
|     06|16 GB Flash Storage|       240| 100   |
|     07|32 GB Flash Storage|       320| 50    |
|     08|64 GB Flash Storage|       430| 25    |
+-------+--------------------+----------+-------+
```

Left and Right DF

1. Join Condition/Expression : `join_expr = order_df.prod_id == product_df.prod_id`
2. Join Type : `joinType = "inner"`

```
order_df.join(product_renamed_df, join_expr, "inner")
```

Always start with Left DF i.e., order_df ^ and passing the right DF ^ product_renamed_df which contains join_expression and join mode which is “inner”. (inner join is default you can skip it.)

- Inner :: find matches left table 1 in all elements in table to :: after matching data it combines both DF as like SQL inner function.

main.py

```
from pyspark.sql import SparkSession

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .appName("SparkJoins")\
        .master("local[3]")\
        .getOrCreate()

    logger = Log4j(spark)

    orders_list = [
        ("01", "02", 350, 1),
        ("01", "04", 580, 1),
        ("01", "07", 320, 2),
        ("02", "03", 450, 1),
        ("02", "06", 220, 1),
        ("03", "01", 195, 1),
        ("04", "09", 270, 3),
        ("04", "08", 410, 2),
```

```
    ("05", "02", 350, 1)
]

order_df = spark.createDataFrame(orders_list).toDF("order_id", "prod_id", "unit_price", "qty")

product_list = [
    ("01", "Scroll Mouse", 250, 20),
    ("02", "Optical Mouse", 350, 20),
    ("03", "Wireless Mouse", 450, 50),
    ("04", "Wireless Keyboard", 580, 50),
    ("05", "Standard Keyboard", 360, 10),
    ("06", "16 GB Flash Storage", 240, 100),
    ("07", "32 GB Flash Storage", 320, 50),
    ("08", "64 GB Flash Storage", 430, 25)
]

product_df = spark.createDataFrame(product_list).toDF("prod_df", "prod_name", "list_price", "qty")

product_df.show()
order_df.show()
```

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

PYSPARK_DRIVER_PYTHON Environment variable for JNotebook

User variables for innk

Variable	Value
HADOOP_HOME	E:\pySpark_soft\hadoop-3.2.2
JAVA_HOME	C:\Program Files\Java\jdk-11
OneDrive	C:\Users\innk\OneDrive
Path	C:\Users\innk\AppData\Local\Programs\Python\Python39\Script...
PYSPARK_DRIVER_PYTHON	C:\Users\innk\anaconda3\python.exe
PYSPARK_PYTHON	C:\Users\innk\AppData\Local\Programs\Python\Python39\pytho...
PYTHONPATH	E:\pySpark_soft\spark_3\python;E:\pySpark_soft\spark_3\python...
SPARK_HOME	E:\pySpark_soft\spark_3

Python in worker has different version 3.9 than that in driver 3.10 :: in PyCharm (NEED TO CHANGE PYSPARK_PYTHON TO 3.10 AND DELETE DRIVER_PYTHON)
Python in worker has different version 3.10 than that in driver 3.9 :: in JNotebook
Note : anaconda running on 3.9 and PyCharm on 3.10

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Result:

```
+-------+-------------------+----------+---+
|prod_df|          prod_name|list_price|qty|
+-------+-------------------+----------+---+
|     01|       Scroll Mouse|       250| 20|
|     02|      Optical Mouse|       350| 20|
|     03|     Wireless Mouse|       450| 50|
|     04|  Wireless Keyboard|       580| 50|
|     05|  Standard Keyboard|       360| 10|
|     06|16 GB Flash Storage|       240|100|
|     07|32 GB Flash Storage|       320| 50|
|     08|64 GB Flash Storage|       430| 25|
+-------+-------------------+----------+---+
```

```
+--------+-------+----------+---+
|order_id|prod_id|unit_price|qty|
+--------+-------+----------+---+
|      01|     02|       350|  1|
|      01|     04|       580|  1|
|      01|     07|       320|  2|
|      02|     03|       450|  1|
|      02|     06|       220|  1|
|      03|     01|       195|  1|
|      04|     09|       270|  3|
|      04|     08|       410|  2|
|      05|     02|       350|  1|
+--------+-------+----------+---+
```

prod_df (* 1st column header is prod_id) oreder_df

Performing Inner Join using innerExpression on above prod_df and order_df

```
join_exp = order_df.prod_id == product_df.prod_id

order_df.join(product_df, join_exp, "inner")\
        .show()
```

```
+--------+-------+----------+---+-------+------------------+----------+---+
|order_id|prod_id|unit_price|qty|prod_id|         prod_name|list_price|qty|
+--------+-------+----------+---+-------+------------------+----------+---+
|      03|     01|       195|  1|     01|       Scroll Mouse|       250| 20|
|      01|     02|       350|  1|     02|      Optical Mouse|       350| 20|
|      05|     02|       350|  1|     02|      Optical Mouse|       350| 20|
|      02|     03|       450|  1|     03|     Wireless Mouse|       450| 50|
|      01|     04|       580|  1|     04|  Wireless Keyboard|       580| 50|
|      02|     06|       220|  1|     06|16 GB Flash Storage|       240|100|
|      01|     07|       320|  2|     07|32 GB Flash Storage|       320| 50|
|      04|     08|       410|  2|     08|64 GB Flash Storage|       430| 25|
+--------+-------+----------+---+-------+------------------+----------+---+
```

- ## Column name ambiguity

```
order_df.join(product_df, join_exp, "inner")\
    .select("qty")\
    .show()
```

```
pyspark.sql.utils.AnalysisException: Reference 'qty' is ambiguous, could be: qty, qty.
```

We got an error for the selected column name “qty”. because spark doesn’t have ambiguity to differ same name column in two different DF. But, when we use select “*” it shows columns with same names because the spark have its own internal ids that will differ these same column name by that ids.

```
order_df.join(product_df, join_exp, "inner")\
    .select("*")\
    .show()
```

```
+--------+-------+----------+---+-------+------------------+----------+---+
|order_id|prod_id|unit_price|qty|prod_id|         prod_name|list_price|qty|
+--------+-------+----------+---+-------+------------------+----------+---+
|      03|     01|       195|  1|     01|       Scroll Mouse|       250| 20|
|      01|     02|       350|  1|     02|      Optical Mouse|       350| 20|
|      05|     02|       350|  1|     02|      Optical Mouse|       350| 20|
|      02|     03|       450|  1|     03|     Wireless Mouse|       450| 50|
|      01|     04|       580|  1|     04|  Wireless Keyboard|       580| 50|
|      02|     06|       220|  1|     06|16 GB Flash Storage|       240|100|
|      01|     07|       320|  2|     07|32 GB Flash Storage|       320| 50|
|      04|     08|       410|  2|     08|64 GB Flash Storage|       430| 25|
+--------+-------+----------+---+-------+------------------+----------+---+
```

We have to take care like rename that same column names before execution.

join_exp = order_df.prod_id == product_df.prod_id

product_renamed_df = product_df.withColumnRenamed("qty", "reorder_qty")
order_df.join(product_renamed_df, join_exp, "inner")\
.drop(product_renamed_df.prod_id)\ # we also have same prod_id colimn in both DF so drop.
.select("prod_id","order_id", "prod_name", "unit_price", "qty")\
.show()

```
+-------+--------+------------------+----------+---+
|prod_id|order_id|         prod_name|unit_price|qty|
+-------+--------+------------------+----------+---+
|     01|      03|      Scroll Mouse|       195|  1|
|     02|      01|     Optical Mouse|       350|  1|
|     02|      05|     Optical Mouse|       350|  1|
|     03|      02|    Wireless Mouse|       450|  1|
|     04|      01| Wireless Keyboard|       580|  1|
|     06|      02|16 GB Flash Storage|       220|  1|
|     07|      01|32 GB Flash Storage|       320|  2|
|     08|      04|64 GB Flash Storage|       410|  2|
+-------+--------+------------------+----------+---+
```

- **Outer Joins in DF**
  - Outer joins are joins that return matched values and unmatched values from either or both tables.

    join_exp = order_df.prod_id == product_df.prod_id

    product_renamed_df = product_df.withColumnRenamed("qty", "reorder_qty")
    order_df.join(product_renamed_df, join_exp, "outer")\
    .drop(product_renamed_df.prod_id)\
    .select("*")\
    .sort("order_id")\
    .show()

```
+--------+-------+----------+----+------------------+----------+-----------+
|order_id|prod_id|unit_price| qty|         prod_name|list_price|reorder_qty|
+--------+-------+----------+----+------------------+----------+-----------+
|    null|   null|      null|null|  Standard Keyboard|       360|         10|
|      01|     04|       580|   1|  Wireless Keyboard|       580|         50|
|      01|     02|       350|   1|      Optical Mouse|       350|         20|
|      01|     07|       320|   2|32 GB Flash Storage|       320|         50|
|      02|     03|       450|   1|     Wireless Mouse|       450|         50|
|      02|     06|       220|   1|16 GB Flash Storage|       240|        100|
|      03|     01|       195|   1|       Scroll Mouse|       250|         20|
|      04|     08|       410|   2|64 GB Flash Storage|       430|         25|
|      04|     09|       270|   3|               null|      null|       null|
|      05|     02|       350|   1|      Optical Mouse|       350|         20|
+--------+-------+----------+----+------------------+----------+-----------+
```

FULL OUTER JOIN

1. Outer Join – Full Outer
2. Left Join – Left Outer
3. Right Join – Right Outer

```
product_renamed_df = product_df.withColumnRenamed("qty", "reorder_qty")
order_df.join(product_renamed_df, join_exp, "left")\
    .drop(product_renamed_df.prod_id)\
    .select("order_id","prod_id", "prod_name", "unit_price", "list_price", "qty")\
    .sort("order_id")\
    .show()
```

```
+--------+-------+-------------------+----------+----------+---+
|order_id|prod_id|          prod_name|unit_price|list_price|qty|
+--------+-------+-------------------+----------+----------+---+
|      01|     07|32 GB Flash Storage|       320|       320|  2|
|      01|     04|  Wireless Keyboard|       580|       580|  1|
|      01|     02|      Optical Mouse|       350|       350|  1|
|      02|     03|     Wireless Mouse|       450|       450|  1|
|      02|     06|16 GB Flash Storage|       220|       240|  1|
|      03|     01|       Scroll Mouse|       195|       250|  1|
|      04|     09|               null|       270|      null|  3|
|      04|     08|64 GB Flash Storage|       410|       430|  2|
|      05|     02|      Optical Mouse|       350|       350|  1|
+--------+-------+-------------------+----------+----------+---+
```

LEFT OUTER JOIN

```
join_exp = order_df.prod_id == product_df.prod_id

product_renamed_df = product_df.withColumnRenamed("qty", "reorder_qty")
order_df.join(product_renamed_df, join_exp, "left")\
    .drop(product_renamed_df.prod_id)\
    .select("order_id","prod_id", "prod_name", "unit_price", "list_price", "qty")\
    .withColumn("prod_name", expr("coalesce(prod_name, prod_id)"))\
    .withColumn("list_price", expr("coalesce(list_price, unit_price)"))\
    .sort("order_id")\
    .show()
    #coalece takes 1st null value replace with 2nd value
```

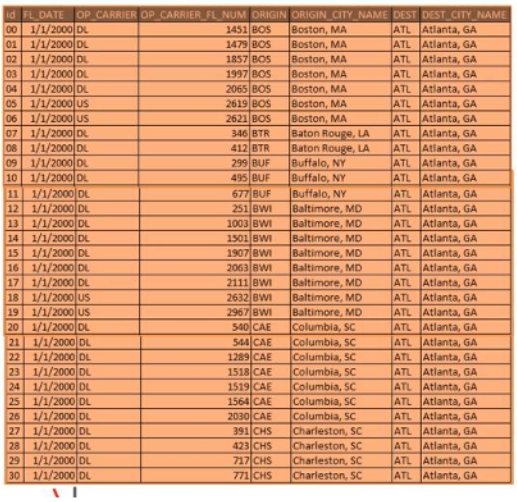
```
+--------+-------+-------------------+----------+----------+---+
|order_id|prod_id|          prod_name|unit_price|list_price|qty|
+--------+-------+-------------------+----------+----------+---+
|      01|     07|32 GB Flash Storage|       320|       320|  2|
|      01|     04|  Wireless Keyboard|       580|       580|  1|
|      01|     02|      Optical Mouse|       350|       350|  1|
|      02|     03|     Wireless Mouse|       450|       450|  1|
|      02|     06|16 GB Flash Storage|       220|       240|  1|
|      03|     01|       Scroll Mouse|       195|       250|  1|
|      04|     09|                 09|       270|       270|  3|
|      04|     08|64 GB Flash Storage|       410|       430|  2|
|      05|     02|      Optical Mouse|       350|       350|  1|
+--------+-------+-------------------+----------+----------+---+
```

Using **coalesce LEFT JOIN**

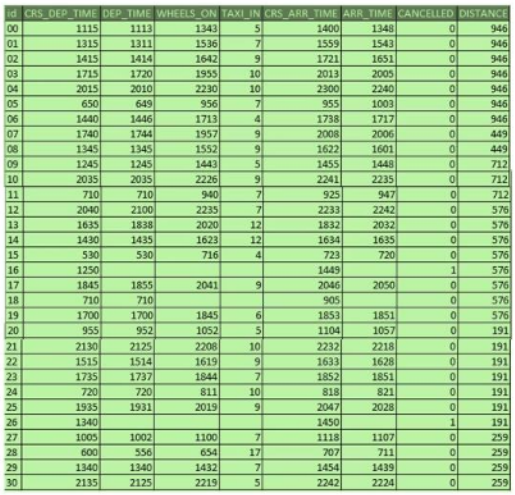
- **Internals of Spark Join and Shuffle**
  - Spark JOIN one of the most commonly causes to slowing down your application.

1. Shuffle Join
2. Broadcast Join

Shuffle JOIN most commonly used.

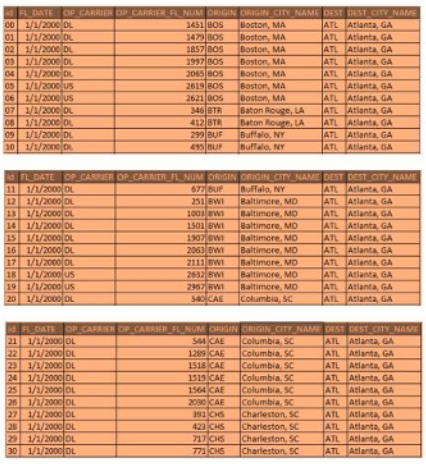


id	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	DEST	DEST_CITY_NAME
00	1/1/2000	DL	1451	BOS	Boston, MA	ATL	Atlanta, GA
01	1/1/2000	DL	1479	BOS	Boston, MA	ATL	Atlanta, GA
02	1/1/2000	DL	1857	BOS	Boston, MA	ATL	Atlanta, GA
03	1/1/2000	DL	1997	BOS	Boston, MA	ATL	Atlanta, GA
04	1/1/2000	DL	2065	BOS	Boston, MA	ATL	Atlanta, GA
05	1/1/2000	US	2619	BOS	Boston, MA	ATL	Atlanta, GA
06	1/1/2000	US	2621	BOS	Boston, MA	ATL	Atlanta, GA
07	1/1/2000	DL	346	BTR	Baton Rouge, LA	ATL	Atlanta, GA
08	1/1/2000	DL	412	BTR	Baton Rouge, LA	ATL	Atlanta, GA
09	1/1/2000	DL	299	BUF	Buffalo, NY	ATL	Atlanta, GA
10	1/1/2000	DL	495	BUF	Buffalo, NY	ATL	Atlanta, GA
11	1/1/2000	DL	677	BUF	Buffalo, NY	ATL	Atlanta, GA
12	1/1/2000	DL	251	BWI	Baltimore, MD	ATL	Atlanta, GA
13	1/1/2000	DL	1003	BWI	Baltimore, MD	ATL	Atlanta, GA
14	1/1/2000	DL	1501	BWI	Baltimore, MD	ATL	Atlanta, GA
15	1/1/2000	DL	1907	BWI	Baltimore, MD	ATL	Atlanta, GA
16	1/1/2000	DL	2063	BWI	Baltimore, MD	ATL	Atlanta, GA
17	1/1/2000	DL	2111	BWI	Baltimore, MD	ATL	Atlanta, GA
18	1/1/2000	US	2632	BWI	Baltimore, MD	ATL	Atlanta, GA
19	1/1/2000	US	2967	BWI	Baltimore, MD	ATL	Atlanta, GA
20	1/1/2000	DL	540	CAE	Columbia, SC	ATL	Atlanta, GA
21	1/1/2000	DL	544	CAE	Columbia, SC	ATL	Atlanta, GA
22	1/1/2000	DL	1289	CAE	Columbia, SC	ATL	Atlanta, GA
23	1/1/2000	DL	1518	CAE	Columbia, SC	ATL	Atlanta, GA
24	1/1/2000	DL	1519	CAE	Columbia, SC	ATL	Atlanta, GA
25	1/1/2000	DL	1564	CAE	Columbia, SC	ATL	Atlanta, GA
26	1/1/2000	DL	2030	CAE	Columbia, SC	ATL	Atlanta, GA
27	1/1/2000	DL	391	CHS	Charleston, SC	ATL	Atlanta, GA
28	1/1/2000	DL	423	CHS	Charleston, SC	ATL	Atlanta, GA
29	1/1/2000	DL	717	CHS	Charleston, SC	ATL	Atlanta, GA
30	1/1/2000	DL	771	CHS	Charleston, SC	ATL	Atlanta, GA



id	CRS_DEP_TIME	DEP_TIME	WHEELS_ON	TAXI_IN	CRS_ARR_TIME	ARR_TIME	CANCELLED	DISTANCE
00	1115	1113	1343	5	1400	1348	0	946
01	1315	1311	1536	7	1559	1543	0	946
02	1415	1414	1642	9	1721	1651	0	946
03	1715	1720	1955	10	2013	2005	0	946
04	2015	2010	2230	10	2300	2240	0	946
05	650	649	956	7	955	1003	0	946
06	1440	1446	1713	4	1738	1717	0	946
07	1740	1744	1957	9	2008	2006	0	449
08	1345	1345	1552	9	1622	1601	0	449
09	1245	1245	1443	5	1455	1448	0	712
10	2035	2035	2226	9	2241	2235	0	712
11	710	710	940	7	925	947	0	712
12	2040	2100	2235	7	2233	2242	0	576
13	1635	1838	2020	12	1832	2032	0	576
14	1430	1435	1623	12	1634	1635	0	576
15	530	530	716	4	723	720	0	576
16	1250				1449		1	576
17	1845	1855	2041	9	2046	2050	0	576
18	710	710			905		0	576
19	1700	1700	1845	6	1853	1851	0	576
20	955	952	1052	5	1104	1057	0	191
21	2130	2125	2208	10	2232	2218	0	191
22	1515	1514	1619	9	1633	1628	0	191
23	1735	1737	1844	7	1852	1851	0	191
24	720	720	811	10	818	821	0	191
25	1935	1931	2019	9	2047	2028	0	191
26	1340				1450		1	191
27	1005	1002	1100	7	1118	1107	0	259
28	600	556	654	17	707	711	0	259
29	1340	1340	1432	7	1454	1439	0	259
30	2135	2125	2219	5	2242	2224	0	259

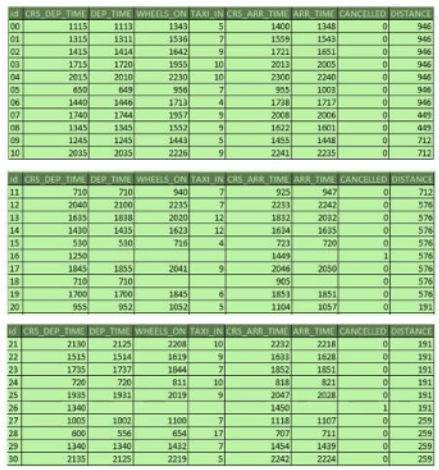
^^^^^^^^^^^^^^^^^^^^^^^^ Partition of above DF^^^^^^^^^^^^^^^^^^^^^^^^



id	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	DEST	DEST_CITY_NAME
00	1/1/2000	DL	1451	BOS	Boston, MA	ATL	Atlanta, GA
01	1/1/2000	DL	1479	BOS	Boston, MA	ATL	Atlanta, GA
02	1/1/2000	DL	1857	BOS	Boston, MA	ATL	Atlanta, GA
03	1/1/2000	DL	1997	BOS	Boston, MA	ATL	Atlanta, GA
04	1/1/2000	DL	2065	BOS	Boston, MA	ATL	Atlanta, GA
05	1/1/2000	US	2619	BOS	Boston, MA	ATL	Atlanta, GA
06	1/1/2000	US	2621	BOS	Boston, MA	ATL	Atlanta, GA
07	1/1/2000	DL	346	BTR	Baton Rouge, LA	ATL	Atlanta, GA
08	1/1/2000	DL	412	BTR	Baton Rouge, LA	ATL	Atlanta, GA
09	1/1/2000	DL	299	BUF	Buffalo, NY	ATL	Atlanta, GA
10	1/1/2000	DL	495	BUF	Buffalo, NY	ATL	Atlanta, GA

id	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	DEST	DEST_CITY_NAME
11	1/1/2000	DL	677	BUF	Buffalo, NY	ATL	Atlanta, GA
12	1/1/2000	DL	251	BWI	Baltimore, MD	ATL	Atlanta, GA
13	1/1/2000	DL	1003	BWI	Baltimore, MD	ATL	Atlanta, GA
14	1/1/2000	DL	1501	BWI	Baltimore, MD	ATL	Atlanta, GA
15	1/1/2000	DL	1907	BWI	Baltimore, MD	ATL	Atlanta, GA
16	1/1/2000	DL	2063	BWI	Baltimore, MD	ATL	Atlanta, GA
17	1/1/2000	DL	2111	BWI	Baltimore, MD	ATL	Atlanta, GA
18	1/1/2000	US	2632	BWI	Baltimore, MD	ATL	Atlanta, GA
19	1/1/2000	US	2967	BWI	Baltimore, MD	ATL	Atlanta, GA
20	1/1/2000	DL	540	CAE	Columbia, SC	ATL	Atlanta, GA

id	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	DEST	DEST_CITY_NAME
21	1/1/2000	DL	544	CAE	Columbia, SC	ATL	Atlanta, GA
22	1/1/2000	DL	1289	CAE	Columbia, SC	ATL	Atlanta, GA
23	1/1/2000	DL	1518	CAE	Columbia, SC	ATL	Atlanta, GA
24	1/1/2000	DL	1519	CAE	Columbia, SC	ATL	Atlanta, GA
25	1/1/2000	DL	1564	CAE	Columbia, SC	ATL	Atlanta, GA
26	1/1/2000	DL	2030	CAE	Columbia, SC	ATL	Atlanta, GA
27	1/1/2000	DL	391	CHS	Charleston, SC	ATL	Atlanta, GA
28	1/1/2000	DL	423	CHS	Charleston, SC	ATL	Atlanta, GA
29	1/1/2000	DL	717	CHS	Charleston, SC	ATL	Atlanta, GA
30	1/1/2000	DL	771	CHS	Charleston, SC	ATL	Atlanta, GA



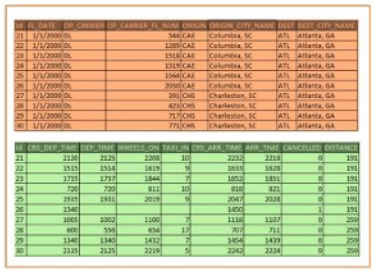
id	CRS_DEP_TIME	DEP_TIME	WHEELS_ON	TAXI_IN	CRS_ARR_TIME	ARR_TIME	CANCELLED	DISTANCE
00	1115	1113	1343	5	1400	1348	0	946
01	1315	1311	1536	7	1559	1543	0	946
02	1415	1414	1642	9	1721	1651	0	946
03	1715	1720	1955	10	2013	2005	0	946
04	2015	2010	2230	10	2300	2240	0	946
05	650	649	956	7	955	1003	0	946
06	1440	1446	1713	4	1738	1717	0	946
07	1740	1744	1957	9	2008	2006	0	449
08	1345	1345	1552	9	1622	1601	0	449
09	1245	1245	1443	5	1455	1448	0	712
10	2035	2035	2226	9	2241	2235	0	712

id	CRS_DEP_TIME	DEP_TIME	WHEELS_ON	TAXI_IN	CRS_ARR_TIME	ARR_TIME	CANCELLED	DISTANCE
11	710	710	940	7	925	947	0	712
12	2040	2100	2235	7	2233	2242	0	576
13	1635	1838	2020	12	1832	2032	0	576
14	1430	1435	1623	12	1634	1635	0	576
15	530	530	716	4	723	720	0	576
16	1250				1449		1	576
17	1845	1855	2041	9	2046	2050	0	576
18	710	710			905		0	576
19	1700	1700	1845	6	1853	1851	0	576
20	955	952	1052	5	1104	1057	0	191

id	CRS_DEP_TIME	DEP_TIME	WHEELS_ON	TAXI_IN	CRS_ARR_TIME	ARR_TIME	CANCELLED	DISTANCE
21	2130	2125	2208	10	2232	2218	0	191
22	1515	1514	1619	9	1633	1628	0	191
23	1735	1737	1844	7	1852	1851	0	191
24	720	720	811	10	818	821	0	191
25	1935	1931	2019	9	2047	2028	0	191
26	1340				1450		1	191
27	1005	1002	1100	7	1118	1107	0	259
28	600	556	654	17	707	711	0	259
29	1340	1340	1432	7	1454	1439	0	259
30	2135	2125	2219	5	2242	2224	0	259

E1

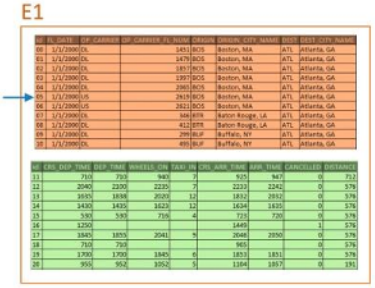
E2



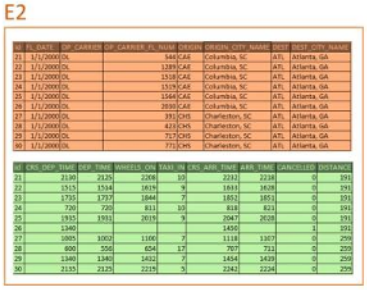
E3

3 Executors E1 E2 E3 | When matching keys are at different executors, joining is no possible unless they are on same executor. Then how? Each executor map these records using joinkeys and send it to exchange

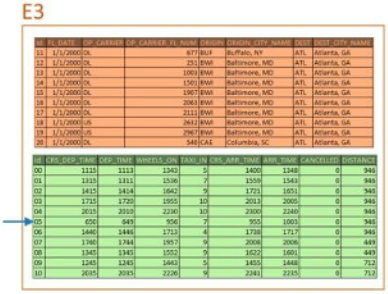
E1



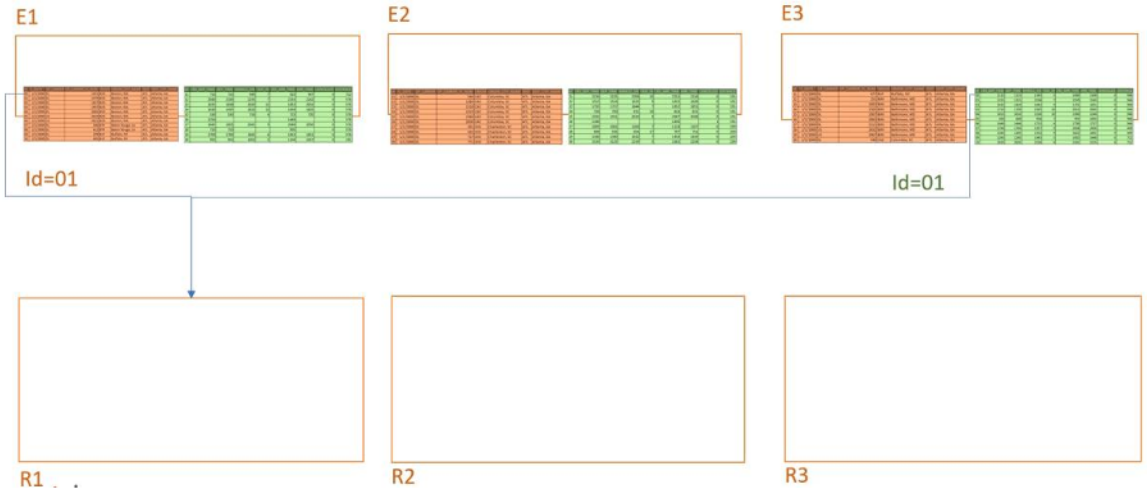
E2



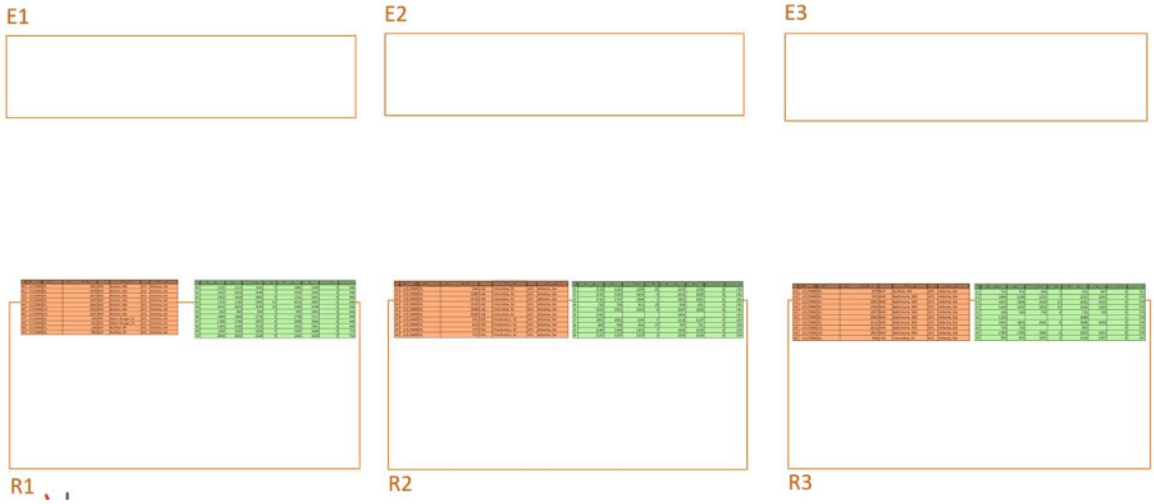
E3



- Map exchange (map reduce) :: Its like buffer at the executor >> map exchange pick these records and send to map reduce exchange >> Reduce exchange going to collect all record for same key.



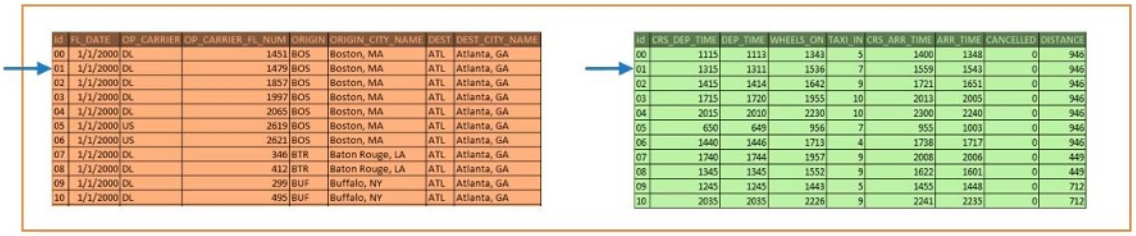
here shuffled for 3 partitions and all partitions are processed in parallel.



0 – 10 keys 20 -30 keys 10-20keys

This is shuffle method is main reason that JOIN reduces the performance of app.

Tunning of JOIN operation Is all about optimization of shuffle operation.

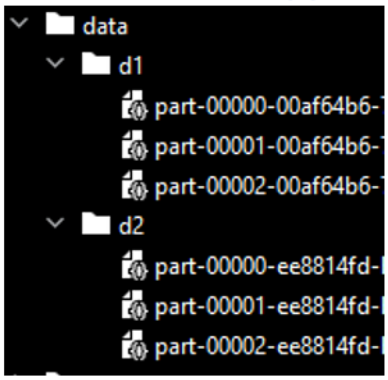


id	FL_DATE	OP_CARRIER	OP_CARRIER_FL_NUM	ORIGIN	ORIGIN_CITY_NAME	DEST	DEST_CITY_NAME
00	1/1/2000	DL	1451	BOS	Boston, MA	ATL	Atlanta, GA
01	1/1/2000	DL	1479	BOS	Boston, MA	ATL	Atlanta, GA
02	1/1/2000	DL	1857	BOS	Boston, MA	ATL	Atlanta, GA
03	1/1/2000	DL	1997	BOS	Boston, MA	ATL	Atlanta, GA
04	1/1/2000	DL	2065	BOS	Boston, MA	ATL	Atlanta, GA
05	1/1/2000	US	2619	BOS	Boston, MA	ATL	Atlanta, GA
06	1/1/2000	US	2621	BOS	Boston, MA	ATL	Atlanta, GA
07	1/1/2000	DL	346	BTR	Baton Rouge, LA	ATL	Atlanta, GA
08	1/1/2000	DL	412	BTR	Baton Rouge, LA	ATL	Atlanta, GA
09	1/1/2000	DL	299	BUF	Buffalo, NY	ATL	Atlanta, GA
10	1/1/2000	DL	495	BUF	Buffalo, NY	ATL	Atlanta, GA

id	CRS_DEP_TIME	DEP_TIME	WHEELS_ON	TAXI_IN	CRS_ARR_TIME	ARR_TIME	CANCELLED	DISTANCE
00	1115	1113	1343	5	1400	1348	0	946
01	1315	1311	1536	7	1559	1543	0	946
02	1415	1414	1642	9	1721	1651	0	946
03	1715	1720	1955	10	2013	2005	0	946
04	2015	2010	2230	10	2300	2240	0	946
05	650	649	956	7	955	1003	0	946
06	1440	1446	1713	4	1738	1717	0	946
07	1740	1744	1957	9	2008	2006	0	449
08	1345	1345	1552	9	1622	1601	0	449
09	1245	1245	1443	5	1455	1448	0	712
10	2035	2035	2226	9	2241	2235	0	712

R1

Shuffle JOIN **main.py**



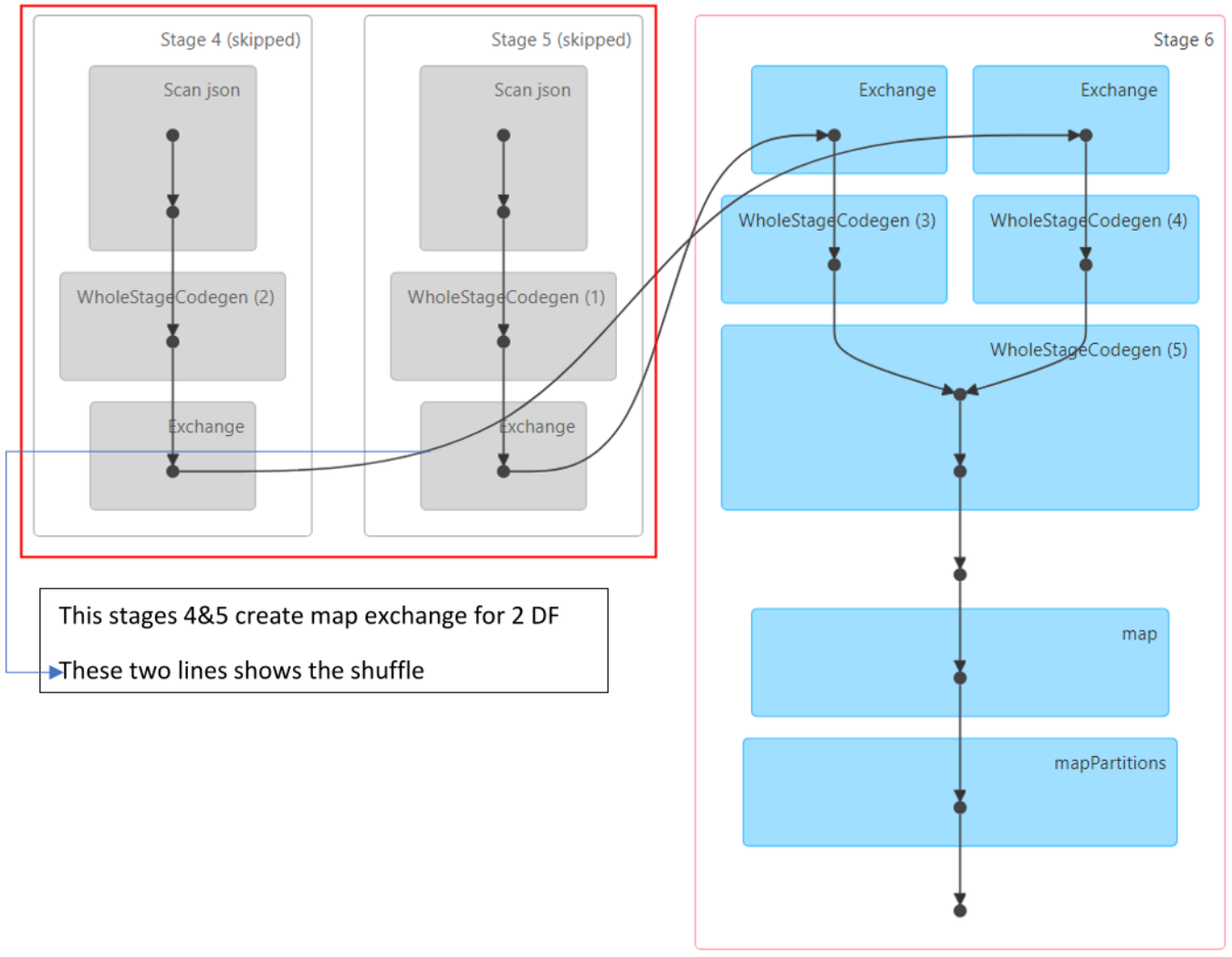
Here we have 2 partitions each containing 3 JSON files.

```
from pyspark.sql import SparkSession
from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[3]")\
        .appName("ShuffleJoin")\
        .getOrCreate()

    logger = Log4j(spark)

    flight_time_df1 = spark.read.json("data/d1/")
    flight_time_df2 = spark.read.json("data/d2/")

    """following conf insures that we having 3 partitions after the shuffle
        which means having 3 exchanges"""
    spark.conf.set("spark.sql.shuffle.partitions", 3)

    join_expr = flight_time_df1.id == flight_time_df2.id
    join_df = flight_time_df1.join(flight_time_df2, join_expr, "inner")

    #Dummy action
    join_df.foreach(lambda f: None)
    input("press a key to stop...")
```

localhost:4040/

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
4	foreach at C:\Users\innk\Desktop\Notes\PySpark\Pycharm\ShuffleJoinDemo\main.py:24 foreach at C:\Users\innk\Desktop\Notes\PySpark\Pycharm\ShuffleJoinDemo\main.py:24	2023/02/28 18:22:50	10 s	1/1 (2 skipped)	3/3 (6 skipped)
3	javaToPython at NativeMethodAccessorImpl.java:0 javaToPython at NativeMethodAccessorImpl.java:0	2023/02/28 18:22:46	4 s	1/1	3/3
2	javaToPython at NativeMethodAccessorImpl.java:0 javaToPython at NativeMethodAccessorImpl.java:0	2023/02/28 18:22:46	3 s	1/1	3/3
1	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2023/02/28 18:22:44	1 s	1/1	3/3
0	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2023/02/28 18:22:41	3 s	1/1	3/3

▾ DAG Visualization



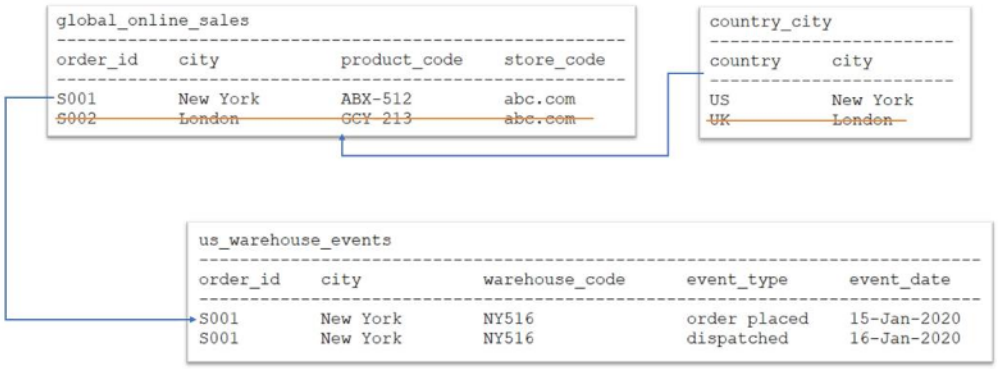
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
6	foreach at C:\Users\innk\Desktop\Notes\PySpark\Pycharm\ShuffleJoinDemo\main.py:24 details	2023/02/28 18:22:50	10 s	3/3			24.4 MiB	

Eliminating this shuffle is best thing in Spark it increases the performance of the app.

- **Optimizing JOINS** ref. 61
  - There are two JOINING scenarios
    1. Large to Large (This always goes for shuffle operations)
    2. Large to Small (This can be take advantage of broadcast JOIN)

1. Don't code like a novice
2. Shuffle partitions and Parallelism
3. Key Distribution

1. Reduce the DF Size before JOIN



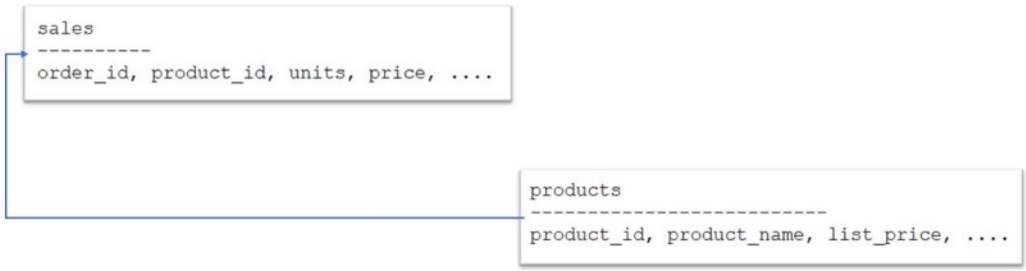
1. Don't code like a novice
2. Shuffle partitions and Parallelism
3. Key Distribution

2.1 Shuffle Partitions
2.2 Number of Executors

>>

What is the maximum possible parallelism?
Executors = 500
Shuffle Partitions = 400
Unique Keys = 200

Example,



200 Unique Products
Max Shuffle Partitions = 200
Max Parallel Tasks = 200

3. Key Distribution

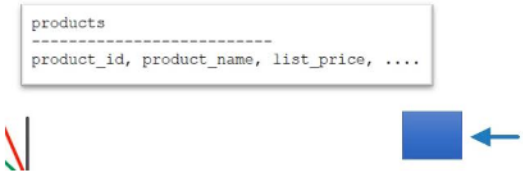
Make partitions as per the performance of the operations

1. Huge Volumes – Filter/Aggregate
2. Parallelism – Shuffles/Executors/Keys
3. Shuffle Distribution – Key Skews

These are the main problems during shuffle joins.

- Large to Small (This can be taken advantage of broadcast JOIN)

  Suppose you have huge DF (sales)



Supposed we used shuffle JOIN

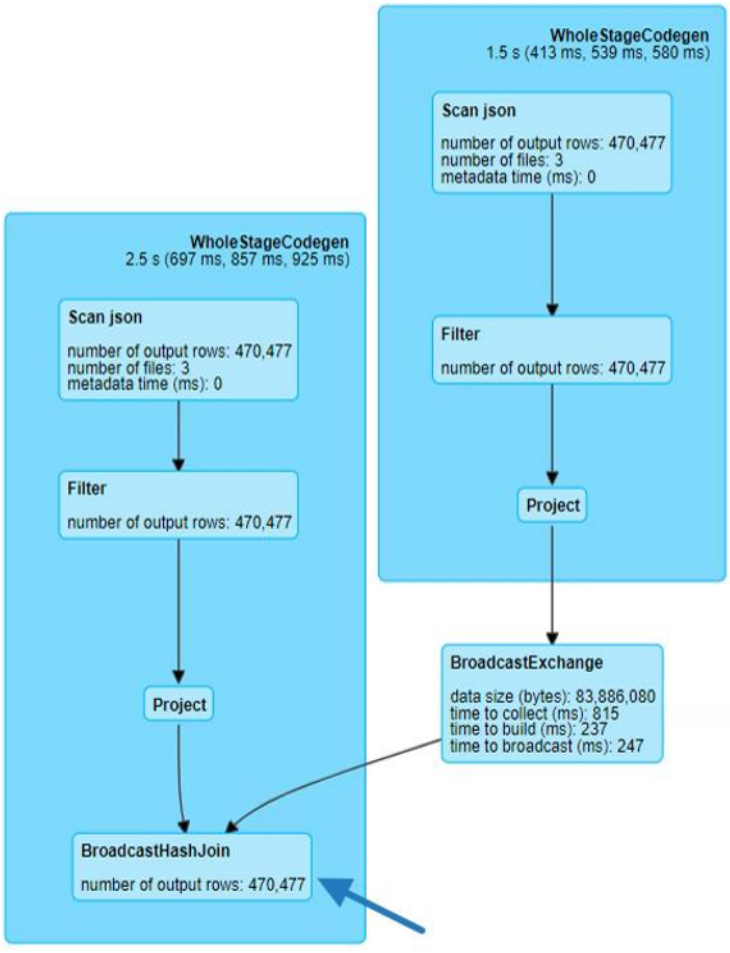
Instead of sending this huge shuffle data to Executors we can dispatch that small size "product" table to all those executors (product table is of 2mb and it will send to 100 executors means now it became 200mb which very small as compared to large which I in GB's product also contains all that column which are in large sales DF) this works without shuffle. Smalls means it can me 1GB or 2GB.

Spark-submit options
--driver-memory
--executor-memory **we know how much memory we have to execute as per that we decide.**

How to broadcast join?

in many cases spark automatically performed broadcasting when large DF and Small DF is present.

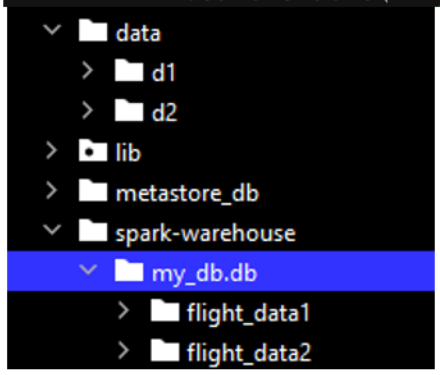
```
from pyspark.sql import SparkSession
from pyspark.sql.functions import broadcast

from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[3]")\
        .appName("ShuffleJoin")\
        .getOrCreate()

    logger = Log4j(spark)

    flight_time_df1 = spark.read.json("data/d1/")
    flight_time_df2 = spark.read.json("data/d2/")

    """following conf insures that we having 3 partitions after the shuffle
        which means having 3 exchanges"""
    spark.conf.set("spark.sql.shuffle.partitions", 3)

    join_expr = flight_time_df1.id == flight_time_df2.id
    join_df = flight_time_df1.join(broadcast(flight_time_df2), join_expr, "inner")

    #Dummy action
    join_df.foreach(lambda f: None)
    input("press a key to stop...")
```



2 (0c634c72-1c6b-412a-9c03-4461de76676b)	broadcast exchange (runId 0c634c72-1c6b-412a-9c03-4461de76676b) $anonfun$withThreadLocalCaptured$1 at FutureTask.java:264

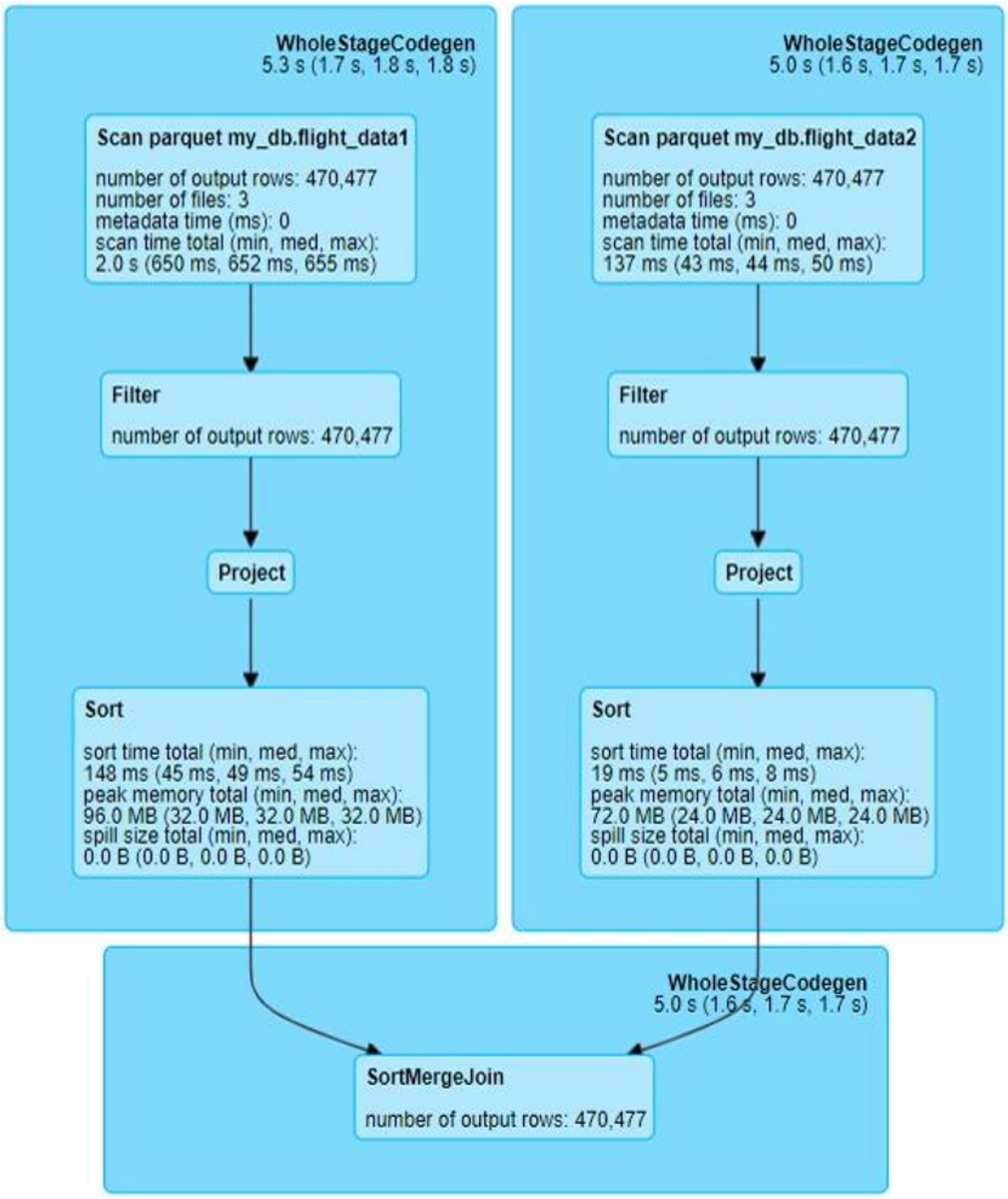
- **Implementing Bucket Join**
  - Plan in advance | The goal is to avoid shuffle.
  - Bucketing is performed only once.
  - Bucketing also need shuffle but only when you create a bucket.
  - When you have bucket you can join DF without Shuffle.
  - There no maximum performance is exist. You have to predict it.
  - By that prediction you can choice buckets as per the compute power we are selecting 3 buckets in following code of creating **Data Base**.

```
from pyspark.sql import SparkSession
from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[3]")\
        .appName("BucketJoin")\
        .enableHiveSupport()\
        .getOrCreate()

    logger = Log4j(spark)

    df1 = spark.read.json("data/d1/")
    df2 = spark.read.json("data/d2/")

    #df1.show()
    #df2.show()

    spark.sql("CREATE DATABASE IF NOT EXISTS MY_DB")
    spark.sql("USE MY_DB")

    df1.coalesce(1).write \
        .bucketBy(3, "id") \
        .mode("overwrite") \
        .saveAsTable("MY_DB.flight_data1")

    df2.coalesce(1).write \
        .bucketBy(3, "id") \
        .mode("overwrite") \
        .saveAsTable("MY_DB.flight_data2")
```



now we have 2 buckets I parquet format.

```python
from pyspark.sql import SparkSession
from lib.logger import Log4j

if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .master("local[3]")\
        .appName("BucketJoin")\
        .enableHiveSupport()\
        .getOrCreate()

    logger = Log4j(spark)

    df1 = spark.read.json("data/d1/")
    df2 = spark.read.json("data/d2/")

    #df1.show()
    #df2.show()

    """spark.sql("CREATE DATABASE IF NOT EXISTS MY_DB")
    spark.sql("USE MY_DB")

    df1.coalesce(1).write \
        .bucketBy(3, "id") \
        .mode("overwrite") \
        .saveAsTable("MY_DB.flight_data1")

    df2.coalesce(1).write \
        .bucketBy(3, "id") \
        .mode("overwrite") \
        .saveAsTable("MY_DB.flight_data2")"""

    df3 = spark.read.table("MY_DB.flight_data1")
    df4 = spark.read.table("MY_DB.flight_data2")

    spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1) # tables are small so the spark automatically broadcast it so by this line of code we disable broadcast.
    join_expr = df3.id == df4.id
    join_df = df3.join(df4, join_expr, "inner")

    join_df.collect()
    input("press a key to stop...")
```



Here you can see we performed join with out Shuffle with the help of bucket.

Its possible only when you preplanned and understanding the data and also understand what and how to use it.