pyspark-tutorial

February 15, 2024

[1]:
```
!pip install pyspark
```

```
Collecting pyspark
  Downloading pyspark-3.5.0.tar.gz (316.9 MB)
                             316.9/316.9
MB 3.0 MB/s eta 0:00:00
  Preparing metadata (setup.py) … done
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.10/dist-
packages (from pyspark) (0.10.9.7)
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) … done
  Created wheel for pyspark: filename=pyspark-3.5.0-py2.py3-none-any.whl
size=317425345
sha256=10666f8a2c9639fa8447d110297ab653b0f15d94dbd6ad2213b9eb2d624a83de
  Stored in directory: /root/.cache/pip/wheels/41/4e/10/c2cf2467f71c678cfc8a6b9a
c9241e5e44a01940da8fbb17fc
Successfully built pyspark
Installing collected packages: pyspark
Successfully installed pyspark-3.5.0
```

[2]:
```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("pyspark").getOrCreate()
```

[3]:
```
type(spark)
```

[3]:
```
pyspark.sql.session.SparkSession
```

0.0.1 Help Function

[ ]:
```
help(spark.createDataFrame)
```

0.0.2 Read csv data in dataframe

[ ]:
```
df = spark.read.csv(path= 'dbfs:/FileStore/data/Employees1.csv', header=True)
df = display(df)
df.printSchema()
```

[ ]:
```
df = spark.read.format('csv').option(key='header', value=True).load(path='dbfs:␣
↪/FileStore/data/Employees1.csv')
display(df)
df.printSchema()
```

0.0.3 Read json data in dataframe

[ ]:
```
# way:1
df.spark.read.json ('dbfs:/FileStore/data/emps.json', multiline = True)
df.spark.read.json (['dbfs:/FileStore/data/emps.json','dbfs:/FileStore/data/
↪emps.json'], multiline = True) # read multiple json files

df.printSchema()
df.show()
```

0.0.4 Read Parquet file into dataframe

[ ]:
```
df = spark.read.parquet('dbfs:/FileStore/data/userdata1.parquet')
display(df)
```

0.0.5 Write dataframe into csv

1. How to read wite dataframe into csv files using pyspark
2. Option available while saving csv files
3. Saving Modes
    - append: Append contents of this class DataFrame to existing data.
    - overwrite: Overwrite existing data.
    - ignore: Silently ignore this operation if data already exists.
    - error or errorifexists (default case): Throw an exception if data already exists.

[ ]:
```
data = [(1, 'Ashutosh'), (2, 'Varsh')]
schema= ['id', 'name']

df = spark.createDataFrame(data=data, schema=schema)

df.write.csv(path='dbfs: /tmp/emps', header=True, mode="erroe")

display(spark.read.csv(path='dbfs: /tmp/emps', header=True))
```

```
DataFrame[id: bigint, name: string]
```

0.0.6 Write dataframe into json

```
[ ]: data = [(1, 'Ashutosh'), (2, 'Varsh')]
schema= ['id', 'name']

df = spark.createDataFrame(data=data, schema=schema)

df.write.json(path='dbfs: /tmp/emps', header=True, mode="erroe")

display(spark.read.json(path='dbfs: /tmp/emps', header=True))
```

```
[ ]: from re import MULTILINE
# way: 2
df = spark.read.format('org.apache.spark.sql.json').load('dbfs:/FileStore/data/
 ↪emps.json', multiline = True)

df.printSchema()
df.show()
```

0.0.7 Write dataframe into Parquet file

```
[ ]: data = [(1, 'Ashutosh'), (2, 'Varsh')]
schema= ['id', 'name']

df = spark.createDataFrame(data=data, schema=schema)

df.write.parquet("/folder_location", mode='error')
```

0.0.8 show() function

```
[ ]: data = [(1,␣
 ↪'dfkdjshjkdbdjfbgdbfgfbdgjkbfdjkgbfskdjbgsfjkdbgjskdbgkjsdbfgjsbdfgb'),
        (2, 'dlfnd;sjhfga;dhfgjadfbgjkfdabgadfjbgadrbfguruigbfdubgiufhdg'),
        (3,␣
 ↪'egrfiuegfuiegfuigeifgsdgfuisdgfiugafiuergifugiuafiuagfiugafuigguaig'),
        (4, 'jhdfghdjbfgbhjfgbhgbfdgbfdgbbfdghbshgfdhsghfdglshlhshdfgshgdshkj')]
schema= ['id', 'commments']

df = spark.createDataFrame(data=data, schema=schema)

df.show(truncate = False)

df.show(truncate = 8)

df.show(n=2, truncate = True)
```

```
df.show(n=2, truncate = True, vertical = True)
```

```
+---+-----------------------------------------------------------------------+
|id |commments                                                              |
+---+-----------------------------------------------------------------------+
|1  |dfkdjshjkdbdjfbgdbfgfbdgjkbfdjkgbfskdjbgsfjkdbgjskdbgkjsdbfgjsbdfgb|
|2  |dlfnd;sjhfga;dhfgjadfbgjkfdabgadfjbgadrbfguruigbfdubgiufhdg        |
|3  |egrfiuegfuiegfuigeifgsdgfuisdgfiugafiuergifugiuafiuagfiugafuigguaig|
|4  |jhdfghdjbfgbhjfgbhgbfdgbfdgbbfdghbshgfdhsghfdglshlhshdfgshgdshkj   |
+---+-----------------------------------------------------------------------+

+---+---------+
| id|commments|
+---+---------+
|  1| dfkdj…|
|  2| dlfnd…|
|  3| egrfi…|
|  4| jhdfg…|
+---+---------+

+---+--------------------+
| id|           commments|
+---+--------------------+
|  1|dfkdjshjkdbdjfbgd…|
|  2|dlfnd;sjhfga;dhfg…|
+---+--------------------+
only showing top 2 rows

-RECORD 0-------------------------
 id        | 1
 commments | dfkdjshjkdbdjfbgd…
-RECORD 1-------------------------
 id        | 2
 commments | dlfnd;sjhfga;dhfg…
only showing top 2 rows
```

0.0.9 withColumn()

- PySpark withColumn() is a transformation function of DataFrame which is used to change the value, convert the datatype of an existing column, create a new column, and many more

[ ]:
```
data = [(1,␣
 ↪'dfkdjshjkdbdjfbgdbfgfbdgjkbfdjkgbfskdjbgsfjkdbgjskdbgkjsdbfgjsbdfgb'),
       (2, 'dlfnd;sjhfga;dhfgjadfbgjkfdabgadfjbgadrbfguruigbfdubgiufhdg'),
       (3,␣
 ↪'egrfiuegfuiegfuigeifgsdgfuisdgfiugafiuergifugiuafiuagfiugafuigguaig'),
```

```
        (4, 'jhdfghdjbfgbhjfgbhgbfdgbfdgbbfdghbshgfdhsghfdglshlhshdfgshgdshkj')]
schema= ['id', 'commments']

df = spark.createDataFrame(data=data, schema=schema)
```

[ ]:
```
from pyspark.sql import functions as F

data = [(1, 'Ashutosh', 3000),
        (2, 'Kumbhare', 4000),
        (3, 'Varsh', 5000),
        (4, 'Ajay', 6000)]
schema= ['id', 'name', 'salary']

df = spark.createDataFrame(data=data, schema=schema)

df = df.withColumn('salary', F.col("salary").cast('Integer'))

df.show()
df.printSchema()

df = df.withColumn('salary', F.col("salary") * 2)
df.show()

df = df.withColumn('country', F.lit('India'))
df.show()

df = df.withColumn('copyed_salary', F.col("salary"))
df.show()
```

```
+---+--------+------+
| id|    name|salary|
+---+--------+------+
|  1|Ashutosh|  3000|
|  2|Kumbhare|  4000|
|  3|   Varsh|  5000|
|  4|    Ajay|  6000|
+---+--------+------+

root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- salary: integer (nullable = true)

+---+--------+------+
| id|    name|salary|
+---+--------+------+
|  1|Ashutosh|  6000|
```

```
|  2|Kumbhare|  8000|
|  3|   Varsh| 10000|
|  4|    Ajay| 12000|
+---+--------+------+

+---+--------+------+-------+
| id|    name|salary|country|
+---+--------+------+-------+
|  1|Ashutosh|  6000|  India|
|  2|Kumbhare|  8000|  India|
|  3|   Varsh| 10000|  India|
|  4|    Ajay| 12000|  India|
+---+--------+------+-------+

+---+--------+------+-------+-------------+
| id|    name|salary|country|copyed_salary|
+---+--------+------+-------+-------------+
|  1|Ashutosh|  6000|  India|         6000|
|  2|Kumbhare|  8000|  India|         8000|
|  3|   Varsh| 10000|  India|        10000|
|  4|    Ajay| 12000|  India|        12000|
+---+--------+------+-------+-------------+
```

0.0.10 withColumnRenamed()

- PySpark withColumnRenamed() is a transformation function of DataFrame which is used to change existing column name in dataframe.

[ ]:
```
data = [(1, 'Ashutosh', 3000),
        (2, 'Kumbhare', 4000),
        (3, 'Varsh', 5000),
        (4, 'Ajay', 6000)]
schema= ['id', 'name', 'salary']

df = spark.createDataFrame(data=data, schema=schema)

df = df.withColumnRenamed('salary', 'Income')
df.show()
```

```
+---+--------+------+
| id|    name|Income|
+---+--------+------+
|  1|Ashutosh|  3000|
|  2|Kumbhare|  4000|
|  3|   Varsh|  5000|
|  4|    Ajay|  6000|
+---+--------+------+
```

0.0.11 StructType() & StructField()

- PySpark StructType & StructField classes are used to programmatically specify the schema to the DataFrame and create complex columns like nested struct, array, and map columns.
- StructType is a collection of StructField's

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql import types as T

data = [(1, 'Ashutosh', 3000),
        (2, 'Kumbhare', 4000),
        (3, 'Varsh', 5000),
        (4, 'Ajay', 6000)]

schema = T.StructType([
                      T.StructField(name='id', dataType=T.IntegerType()),
                      T.StructField(name='name', dataType=T.StringType()),
                      T.StructField(name='salary', dataType=T.IntegerType())
                      ])


df = spark.createDataFrame(data=data, schema=schema)

df.show()
df.printSchema()
```

```
+---+--------+------+
| id|    name|salary|
+---+--------+------+
|  1|Ashutosh|  3000|
|  2|Kumbhare|  4000|
|  3|   Varsh|  5000|
|  4|    Ajay|  6000|
+---+--------+------+

root
 |-- id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- salary: integer (nullable = true)
```

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql import types as T

data = [(1, ('Ashutosh','Kumbhare'), 3000),
        (2, ('Ashu','abc'), 4000),
```

```
        (3, ('Varsh','cde'), 5000),
        (4, ('Ajay','efg'), 6000)]

my_structType = T.StructType(
        [
        T.StructField(name='First_name', dataType=T.StringType()),
        T.StructField(name='Last_name', dataType=T.StringType())
        ]
    )

schema = T.StructType(
                        [
                        T.StructField(name='id', dataType=T.IntegerType()),
                        T.StructField(name='name', dataType=my_structType),
                        T.StructField(name='salary', dataType=T.IntegerType())
                        ]
        )


df = spark.createDataFrame(data=data, schema=schema)

df.show()
df.printSchema()
```

```
+---+--------------------+------+
| id|                name|salary|
+---+--------------------+------+
|  1|{Ashutosh, Kumbhare}|  3000|
|  2|         {Ashu, abc}|  4000|
|  3|        {Varsh, cde}|  5000|
|  4|         {Ajay, efg}|  6000|
+---+--------------------+------+

root
 |-- id: integer (nullable = true)
 |-- name: struct (nullable = true)
 |    |-- First_name: string (nullable = true)
 |    |-- Last_name: string (nullable = true)
 |-- salary: integer (nullable = true)
```

0.0.12 ArrayType Columns

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql import types as T

data = [('abc', [1,2]), ('mno', [4,5]), ('xyz', [7,8])]
```

```
schema = T.StructType([T.StructField(name='id', dataType= T.StringType()),
                       T.StructField(name='numbers', dataType= T.ArrayType(T.
  ↪IntegerType()))
                      ])

df = spark.createDataFrame(data=data, schema=schema)

df.show()
df.printSchema()

df = df.withColumn('index_zero', F.col('numbers')[0])
# df = df.withColumn('index_zero', df.numbers[0])

df.show()
```

```
+---+-------+
| id|numbers|
+---+-------+
|abc| [1, 2]|
|mno| [4, 5]|
|xyz| [7, 8]|
+---+-------+

root
 |-- id: string (nullable = true)
 |-- numbers: array (nullable = true)
 |    |-- element: integer (containsNull = true)

+---+-------+----------+
| id|numbers|index_zero|
+---+-------+----------+
|abc| [1, 2]|         1|
|mno| [4, 5]|         4|
|xyz| [7, 8]|         7|
+---+-------+----------+
```

[ ]:
```
data = [(1, 'Ashutosh', 3000),
        (2, 'Kumbhare', 4000),
        (3, 'Varsh', 5000),
        (4, 'Ajay', 6000)]

schema = T.StructType([
                      T.StructField(name='id', dataType=T.IntegerType()),
                      T.StructField(name='name', dataType=T.StringType()),
                      T.StructField(name='salary', dataType=T.IntegerType())
```

```
                    ])


df = spark.createDataFrame(data=data, schema=schema)

df = df.withColumn('new_column',
                   F.array(
                       F.col('id'),
                       F.col('salary')
                       )
                   )
df.show()
```

```
+---+--------+------+----------+
| id|    name|salary|new_column|
+---+--------+------+----------+
|  1|Ashutosh|  3000| [1, 3000]|
|  2|Kumbhare|  4000| [2, 4000]|
|  3|   Varsh|  5000| [3, 5000]|
|  4|    Ajay|  6000| [4, 6000]|
+---+--------+------+----------+
```

0.0.13 explode(), split(), array() & array_contains() Functions

- Use explode() function to create a new row for each element in the given array column.
- split() sql function returns an array type after splitting the string column by delimiter.
- Use array() function to create a new array column by merging the data from multiple columns.
- array_contains() sql function is used to check if array column contains a value. Returns null if the array is null, true if the array contains the value, and false otherwise.

1. explode()

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql import types as T

data = [('abc', [1,2]), ('efg', [4,5]), ('xyz', [7,8])]

schema = T.StructType(
    [
        T.StructField('id', T.StringType()),
        T.StructField('numbers', T.ArrayType(T.IntegerType()))
        ]
    )

df = spark.createDataFrame(data,schema)
```

```
df = df.withColumn('explodedCol', F.explode(F.col('numbers')))
df.show()

df = df.withColumn('explodedCol', F.explode(F.col('numbers')))\
     .select(
          'id',
          'explodedCol'
          )
df.show()
```

```
+---+-------+-----------+
| id|numbers|explodedCol|
+---+-------+-----------+
|abc| [1, 2]|          1|
|abc| [1, 2]|          2|
|efg| [4, 5]|          4|
|efg| [4, 5]|          5|
|xyz| [7, 8]|          7|
|xyz| [7, 8]|          8|
+---+-------+-----------+

+---+-----------+
| id|explodedCol|
+---+-----------+
|abc|          1|
|abc|          2|
|abc|          1|
|abc|          2|
|efg|          4|
|efg|          5|
|efg|          4|
|efg|          5|
|xyz|          7|
|xyz|          8|
|xyz|          7|
|xyz|          8|
+---+-----------+
```

2. split()

[ ]:
```
from pyspark.sql import functions as F

data = [(1, 'Ashutosh', 'dothet, azure, sql'), (2, 'Varsh','java, aws, sql')]

schema = ['id', 'name', 'skills']

df = spark.createDataFrame(data, schema)
```

```
df.show()

df = df.withColumn('skills',F.split('skills',','))
df.show()

df.printSchema()
```

```
+---+--------+------------------+
| id|    name|            skills|
+---+--------+------------------+
|  1|Ashutosh|dothet, azure, sql|
|  2|   Varsh|    java, aws, sql|
+---+--------+------------------+

+---+--------+--------------------+
| id|    name|              skills|
+---+--------+--------------------+
|  1|Ashutosh|[dothet,  azure, …|
|  2|   Varsh|  [java,  aws,  sql]|
+---+--------+--------------------+

root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- skills: array (nullable = true)
 |    |-- element: string (containsNull = false)
```

3. array()

[ ]:
```
from pyspark.sql import functions as F

data = [(1, 'Ashutosh', 'dotnet', 'azure'), (2, 'Varsha', 'java', 'aws')]

schema = ['id', 'name', 'primarySkill', 'secondarySkill']

df = spark.createDataFrame(data, schema)
df.show()

df = df.withColumn('skills', F.array(F.col('primarySkill'), F.
 ↪col('secondarySkill')))
df.show()

df.printSchema()
```

```
+---+--------+-----------+-------------+
| id|    name|primarySkill|secondarySkill|
+---+--------+-----------+-------------+
```

```
|  1|Ashutosh|      dotnet|         azure|
|  2|  Varsha|        java|           aws|
+---+--------+------------+--------------+

+---+--------+------------+--------------+---------------+
| id|    name|primarySkill|secondarySkill|         skills|
+---+--------+------------+--------------+---------------+
|  1|Ashutosh|      dotnet|         azure|[dotnet, azure]|
|  2|  Varsha|        java|           aws|    [java, aws]|
+---+--------+------------+--------------+---------------+

root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- primarySkill: string (nullable = true)
 |-- secondarySkill: string (nullable = true)
 |-- skills: array (nullable = false)
 |    |-- element: string (containsNull = true)
```

4. array_contains()

[ ]:
```
from pyspark.sql import functions as F

data = [(1, 'Varsh', ['.net', 'azure']), (2, 'Ashutosh', ['java', 'aws'])]

schema = ['id', 'name', 'skills']

df = spark.createDataFrame (data, schema)

df = df.withColumn ('Has JavaSkill', F.array_contains(F.col('skills'), 'java'))
df.show()

df.printSchema()
```

```
+---+--------+-------------+-------------+
| id|    name|       skills|Has JavaSkill|
+---+--------+-------------+-------------+
|  1|   Varsh|[.net, azure]|        false|
|  2|Ashutosh|  [java, aws]|         true|
+---+--------+-------------+-------------+

root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- skills: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- Has JavaSkill: boolean (nullable = true)
```

0.0.14 MapType column

- PySpark MapType is used to represent map key-value pair similar to python Dictionary (Dict)

[ ]:
```
data = [('Ashutosh', {'hair': 'black', 'eye': 'brown'}), ('varsh', { 'hair':␣
 ↪'black', 'eye': 'blue'})]

schema = ['name', 'properties']

df = spark.createDataFrame (data, schema)

df.show()
df.printSchema()
```

```
+--------+--------------------+
|    name|          properties|
+--------+--------------------+
|Ashutosh|{eye -> brown, ha…|
|   varsh|{eye -> blue, hai…|
+--------+--------------------+

root
 |-- name: string (nullable = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql import types as T

data = [('Ashutosh', { 'hair': 'black', 'eye': 'brown'}), ('varsh', {'hair':␣
 ↪'black', 'eye': 'blue'})]

schema = T.StructType(
    [
        T.StructField('name', T.StringType()),
        T.StructField('properties', T.MapType(T.StringType(), T.StringType()))
        ]
    )

df = spark.createDataFrame(data, schema)

df.show(truncate=False)
df.printSchema()
```

```
+--------+-----------------------------+
|name    |properties                   |
+--------+-----------------------------+
|Ashutosh|{eye -> brown, hair -> black}|
|varsh   |{eye -> blue, hair -> black} |
+--------+-----------------------------+

root
 |-- name: string (nullable = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

Access MapType elements

[ ]:
```
# way 1

df1 = df.withColumn('hair', df.properties['hair'])

df1.show(truncate=False)



# way2

df2 = df.withColumn('eye', df.properties.getItem('eye'))

df2.show(truncate=False)
```

```
+--------+-----------------------------+-----+
|name    |properties                   |hair |
+--------+-----------------------------+-----+
|Ashutosh|{eye -> brown, hair -> black}|black|
|varsh   |{eye -> blue, hair -> black} |black|
+--------+-----------------------------+-----+

+--------+-----------------------------+-----+
|name    |properties                   |eye  |
+--------+-----------------------------+-----+
|Ashutosh|{eye -> brown, hair -> black}|brown|
|varsh   |{eye -> blue, hair -> black} |blue |
+--------+-----------------------------+-----+
```

0.0.15 map_keys(), map_values(), explode()

0.0.16 explode() with MapType

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql import types as T

data = [('Ashutosh', { 'hair': 'black', 'eye': 'brown'}), ('varsh', {'hair':␣
 ↪'black', 'eye': 'blue'})]

schema = T.StructType(
    [
        T.StructField('name', T.StringType()),
        T.StructField('properties', T.MapType(T.StringType(), T.StringType()))
        ]
    )

df = spark.createDataFrame(data, schema)
df.show(truncate=False)

df1 = df.select('name', 'properties', F.explode(df.properties))
df1.show(truncate=False)
```

```
+--------+-----------------------------+
|name    |properties                   |
+--------+-----------------------------+
|Ashutosh|{eye -> brown, hair -> black}|
|varsh   |{eye -> blue, hair -> black} |
+--------+-----------------------------+

+--------+-----------------------------+----+-----+
|name    |properties                   |key |value|
+--------+-----------------------------+----+-----+
|Ashutosh|{eye -> brown, hair -> black}|eye |brown|
|Ashutosh|{eye -> brown, hair -> black}|hair|black|
|varsh   |{eye -> blue, hair -> black} |eye |blue |
|varsh   |{eye -> blue, hair -> black} |hair|black|
+--------+-----------------------------+----+-----+
```

2. map_keys()

[ ]:
```
df1 = df.withColumn('keys', F.map_keys(df.properties))

df1.show(truncate=False)
```

```
+--------+-----------------------------+-----------+
|name    |properties                   |keys       |
+--------+-----------------------------+-----------+
```

```
|Ashutosh|{eye -> brown, hair -> black}|[eye, hair]|
|varsh   |{eye -> blue, hair -> black} |[eye, hair]|
+--------+-----------------------------+-----------+
```

3. map.values()

[ ]:
```
df1 = df.withColumn('keys', F.map_values(df.properties))
df1.show(truncate=False)
```

```
+--------+-----------------------------+--------------+
|name    |properties                   |keys          |
+--------+-----------------------------+--------------+
|Ashutosh|{eye -> brown, hair -> black}|[brown, black]|
|varsh   |{eye -> blue, hair -> black} |[blue, black] |
+--------+-----------------------------+--------------+
```

0.0.17 Row() class

- pyspark.sql.Row which is represented as a record/row in DataFrame, one can create a Row object by using named arguments or create a custom Row like class.

[ ]:
```
from pyspark.sql import Row
from pyspark.sql import functions as F

row = Row(name = 'Ashutosh', salary = 2000)

print(row.name)
```

```
Ashutosh
```

[ ]:
```
row1 = Row(name = 'Ashutosh', salary = 2000)
row2 = Row(name = 'Keahav', salary = 3000)

data = [row1, row2]

df = spark.createDataFrame(data)
df.show()
```

```
+--------+------+
|    name|salary|
+--------+------+
|Ashutosh|  2000|
|  Keahav|  3000|
+--------+------+
```

create nested struct type using Row()

[ ]:
```
from pyspark.sql import Row

data=[Row(name="keashav", prop=Row(hair="black", eye="blue")),␣
 ↪Row(name="Ashutosh", prop=Row(hair="grey", eye="black"))]
df=spark.createDataFrame(data)

df.printSchema()
```

```
root
 |-- name: string (nullable = true)
 |-- prop: struct (nullable = true)
 |    |-- hair: string (nullable = true)
 |    |-- eye: string (nullable = true)
```

0.0.18 Column() Class

- PySpark Column class represents a single Column in a DataFrame. pyspark.sql.Column class provides several functions to work with DataFrame to manipulate the Column values, evaluate the boolean expression to filter rows, retrieve a value or part of a value from a DataFrame column
- One of the simplest ways to create a Column class object is by using PySpark lit() SQL function

[ ]:
```
from pyspark.sql import functions as F

col1 = F.lit("zero")

print(type(col1))
```

```
<class 'pyspark.sql.column.Column'>
```

Access columns in multiple ways

[ ]:
```
from pyspark.sql import functions as F

data = [(1, 'Ashutosh', 3000),
        (2, 'Kumbhare', 4000),
        (3, 'Varsh', 5000),
        (4, 'Ajay', 6000)]
schema= ['id', 'name', 'salary']

df = spark.createDataFrame(data=data, schema=schema)

df.select(df.salary).show()
df.select(df['salary']).show()
df.select(F.col('salary')).show()
```

```
+------+
|salary|
+------+
|  3000|
|  4000|
|  5000|
|  6000|
+------+

+------+
|salary|
+------+
|  3000|
|  4000|
|  5000|
|  6000|
+------+

+------+
|salary|
+------+
|  3000|
|  4000|
|  5000|
|  6000|
+------+
```

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql import types as T

data = [(1, ('Ashutosh','Kumbhare'), 3000),
        (2, ('Ashu','abc'), 4000),
        (3, ('Varsh','cde'), 5000),
        (4, ('Ajay','efg'), 6000)]

my_structType = T.StructType(
        [
        T.StructField(name='First_name', dataType=T.StringType()),
        T.StructField(name='Last_name', dataType=T.StringType())
        ]
    )

schema = T.StructType(
                      [
                      T.StructField(name='id', dataType=T.IntegerType()),
                      T.StructField(name='name', dataType=my_structType),
```

```
                        T.StructField(name='salary', dataType=T.IntegerType())
                        ]
        )


df = spark.createDataFrame(data=data, schema=schema)

df.select(df.name.First_name).show()
df.select(df['name.First_name']).show()
df.select(F.col('name.First_name')).show()
```

```
+---------------+
|name.First_name|
+---------------+
|       Ashutosh|
|           Ashu|
|          Varsh|
|           Ajay|
+---------------+

+----------+
|First_name|
+----------+
|  Ashutosh|
|      Ashu|
|     Varsh|
|      Ajay|
+----------+

+----------+
|First_name|
+----------+
|  Ashutosh|
|      Ashu|
|     Varsh|
|      Ajay|
+----------+
```

0.0.19 when() & otherwise()

- It is similar to SQL Case When, executes sequence of expressions until it matches the condition and returns a value when match.

[ ]:
```
from pyspark.sql import functions as F

data = [(1, 'Ashu', 'M', 2000), (2, 'Varsha', 'F', 3000), (3, 'Bhushan', '',␣
↪2000)]
```

```
schema = ['id', 'name', 'gender', 'salary']

df = spark.createDataFrame (data, schema)

df.show()

df1 = df.select(
    df.id,
    df.name,
    F.when(condition=df.gender == 'M', value='Male')
    .when(condition=df.gender == 'F',value='FeMale')
    .otherwise ('unknown').alias('gender')
    )

df1.show()
```

```
+---+-------+------+------+
| id|   name|gender|salary|
+---+-------+------+------+
|  1|   Ashu|     M|  2000|
|  2| Varsha|     F|  3000|
|  3|Bhushan|      |  2000|
+---+-------+------+------+

+---+-------+-------+
| id|   name| gender|
+---+-------+-------+
|  1|   Ashu|   Male|
|  2| Varsha| FeMale|
|  3|Bhushan|unknown|
+---+-------+-------+
```

0.0.20 alias(), asc(), desc(), cast() & like()

1. alies()

[ ]:
```
data = [(1, 'Ashu', 'M',2000), (2, 'Keshav', 'M', 4000), (3, 'Varsha', 'F',␣
↪3000)]

schema = ['id', 'name', 'gender', 'salary']

df = spark.createDataFrame(data, schema)

df.select(
    df.id.alias('emp_id'),
    df.name.alias ('emp_name')
```

```
    )

df.show()
```

```
+---+------+------+------+
| id|  name|gender|salary|
+---+------+------+------+
|  1|  Ashu|     M|  2000|
|  2|Keshav|     M|  4000|
|  3|Varsha|     F|  3000|
+---+------+------+------+
```

0.0.21 asc(), desc()

[ ]:
```
data = [(1, 'Ashutosh', 'M', 2000), (2, 'keshav', 'M', 4000), (3, 'Bhushan',␣
 ↪'F', 3000)]

schema = ['id', 'name', 'gender', 'salary']

spark.createDataFrame(data, schema)

df.sort(df.name.desc()).show()
```

```
+---+--------+------+------+
| id|    name|gender|salary|
+---+--------+------+------+
|  3|  Sonali|     F|  3000|
|  2| Bhushan|     M|  4000|
|  1|Ashutosh|     M|  2000|
+---+--------+------+------+
```

cast()

[ ]:
```
data = [(1, 'Ashutosh', 'M', 2000), (2, 'Bhushan', 'M', 4000), (3, 'Sonali',␣
 ↪'F', 3000)]

schema = ['id', 'name', 'gender', 'salary']

df = spark.createDataFrame(data, schema)
df.printSchema()

df1 = df.select(df.salary.cast('int'))
df1.printSchema()
```

```
root
 |-- id: long (nullable = true)
```

```
 |-- name: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: long (nullable = true)

root
 |-- salary: integer (nullable = true)
```

like()

[ ]:
```
data = [(1, 'Ashutosh', 'M', 2000), (2, 'Keshav', 'M', 4000), (3, 'Varsha',␣
 ↪'F', 3000)]

schema = ['id', 'name', 'gender', 'salary']

df = spark.createDataFrame (data, schema)

df.filter(df.name.like ('a%')).show()

# Like functiuon is case sensitive.
df.filter(df.name.like ('A%')).show()
```

```
+---+----+------+------+
| id|name|gender|salary|
+---+----+------+------+
+---+----+------+------+

+---+--------+------+------+
| id|    name|gender|salary|
+---+--------+------+------+
|  1|Ashutosh|     M|  2000|
+---+--------+------+------+
```

0.0.22 filter()

- PySpark filter() function is used to filter the rows from DataFrame based on the given condition or SQL expression.

0.0.23 where()

- You can also use where() clause instead of the filter() if you are coming from an SQL background, both these functions operate exactly the same.

[ ]:
```
data = [(1, 'Ashutosh', 'M', 2000), (2, 'Keshav', 'M', 4000), (3, 'Varsha',␣
 ↪'F',3000)]

schema = ['id', 'name', 'gender', 'salary']
```

```
df = spark.createDataFrame (data, schema)

# where function
df.where(df.gender == 'M').show()

# filter function
df.filter(df.gender =='M').show()
```

```
+---+--------+------+------+
| id|    name|gender|salary|
+---+--------+------+------+
|  1|Ashutosh|     M|  2000|
|  2|  Keshav|     M|  4000|
+---+--------+------+------+

+---+--------+------+------+
| id|    name|gender|salary|
+---+--------+------+------+
|  1|Ashutosh|     M|  2000|
|  2|  Keshav|     M|  4000|
+---+--------+------+------+
```

0.0.24 distinct()

- PySpark distinct() function is used to remove the duplicate rows (all columns)

0.0.25 dropDuplicates()

- dropDuplicates() is used to drop rows based on selected (one or multiple) columns.

So basically, using these functions we can get distinct rows

[ ]:
```
data = [(1, 'Ashutosh', 'M', 2000), (2, 'Keshav', 'M', 4000), (2, 'Akash',␣
 ↪'M',4000), (3, 'Varsha', 'F', 3000)]

schema = ['id', 'name', 'gender', 'salary']

df = spark.createDataFrame(data, schema)

df.distinct().show()

df.dropDuplicates ().show()

df.dropDuplicates (['gender']).show()
```

```
+---+--------+------+------+
| id|    name|gender|salary|
+---+--------+------+------+
```

```
|  2|  Keshav|     M|  4000|
|  1|Ashutosh|     M|  2000|
|  2|   Akash|     M|  4000|
|  3|  Varsha|     F|  3000|
+---+--------+------+------+

+---+--------+------+------+
| id|    name|gender|salary|
+---+--------+------+------+
|  2|  Keshav|     M|  4000|
|  1|Ashutosh|     M|  2000|
|  2|   Akash|     M|  4000|
|  3|  Varsha|     F|  3000|
+---+--------+------+------+

+---+--------+------+------+
| id|    name|gender|salary|
+---+--------+------+------+
|  3|  Varsha|     F|  3000|
|  1|Ashutosh|     M|  2000|
+---+--------+------+------+
```

0.0.26 orderBy() & sort()

- You can use either sort() or orderBy() function of PySpark DataFrame to sort DataFrame by ascending or descending order based on single or multiple columns.
- By default, sorting will happen in ascending order. We can explicitly mention ascending or descending using asc(), desc() functions.

[ ]:
```
data = [(1, 'Ashutosh', 'M',2000, 'IT'), (2, 'Keshav', 'M',4000, 'HR'), (3,␣
 ↪'Varsha', 'F',3000, 'Payroll'), (4, 'Akash', 'M',3000, 'HR')]

schema = ['id', 'name', 'gender', 'salary', 'dep']

df = spark.createDataFrame(data, schema)

# sort function
df.sort('dep', 'salary').show()
df.sort(df.dep.desc(),df.salary.desc()).show()

# orderBy
df.orderBy(df.dep.desc(),df.salary.asc()).show()
df.orderBy('dep', 'salary').show()
```

```
+---+--------+------+------+-------+
| id|    name|gender|salary|    dep|
+---+--------+------+------+-------+
```

```
|  4|   Akash|     M|  3000|     HR|
|  2|  Keshav|     M|  4000|     HR|
|  1|Ashutosh|     M|  2000|     IT|
|  3|  Varsha|     F|  3000|Payroll|
+---+--------+------+------+-------+

+---+--------+------+------+-------+
| id|    name|gender|salary|    dep|
+---+--------+------+------+-------+
|  3|  Varsha|     F|  3000|Payroll|
|  1|Ashutosh|     M|  2000|     IT|
|  2|  Keshav|     M|  4000|     HR|
|  4|   Akash|     M|  3000|     HR|
+---+--------+------+------+-------+

+---+--------+------+------+-------+
| id|    name|gender|salary|    dep|
+---+--------+------+------+-------+
|  3|  Varsha|     F|  3000|Payroll|
|  1|Ashutosh|     M|  2000|     IT|
|  4|   Akash|     M|  3000|     HR|
|  2|  Keshav|     M|  4000|     HR|
+---+--------+------+------+-------+

+---+--------+------+------+-------+
| id|    name|gender|salary|    dep|
+---+--------+------+------+-------+
|  4|   Akash|     M|  3000|     HR|
|  2|  Keshav|     M|  4000|     HR|
|  1|Ashutosh|     M|  2000|     IT|
|  3|  Varsha|     F|  3000|Payroll|
+---+--------+------+------+-------+
```

0.0.27 union() & unionAll()

- union() and unionAll() transformations are used to merge two or more DataFrame's of the same schema or structure.
- union() & unionAll() method merges two DataFrames and returns the new DataFrame with all rows from two Dataframes regardless of duplicate data. To remove duplicates use distinct() function

[ ]:
```
data = [(1, 'Ashutosh', 'M', 2000, 'IT'), (2, 'Keshav', 'M',4000, 'HR'), (3,␣
 ↪'Varsha', 'F', 3000, 'Payroll')]

schema = ['id', 'name', 'gender', 'salary', 'dep']
```

```
df1 = spark.createDataFrame(data, schema)

df2 = spark.createDataFrame(data, schema)

df1.show()

df2.show()

df1.unionAll(df2).show()

df1.union(df2).show()
```

```
+---+--------+------+------+-------+
| id|    name|gender|salary|    dep|
+---+--------+------+------+-------+
|  1|Ashutosh|     M|  2000|     IT|
|  2|  Keshav|     M|  4000|     HR|
|  3|  Varsha|     F|  3000|Payroll|
+---+--------+------+------+-------+

+---+--------+------+------+-------+
| id|    name|gender|salary|    dep|
+---+--------+------+------+-------+
|  1|Ashutosh|     M|  2000|     IT|
|  2|  Keshav|     M|  4000|     HR|
|  3|  Varsha|     F|  3000|Payroll|
+---+--------+------+------+-------+

+---+--------+------+------+-------+
| id|    name|gender|salary|    dep|
+---+--------+------+------+-------+
|  1|Ashutosh|     M|  2000|     IT|
|  2|  Keshav|     M|  4000|     HR|
|  3|  Varsha|     F|  3000|Payroll|
|  1|Ashutosh|     M|  2000|     IT|
|  2|  Keshav|     M|  4000|     HR|
|  3|  Varsha|     F|  3000|Payroll|
+---+--------+------+------+-------+

+---+--------+------+------+-------+
| id|    name|gender|salary|    dep|
+---+--------+------+------+-------+
|  1|Ashutosh|     M|  2000|     IT|
|  2|  Keshav|     M|  4000|     HR|
|  3|  Varsha|     F|  3000|Payroll|
|  1|Ashutosh|     M|  2000|     IT|
```

```
|  2|  Keshav|     M|  4000|     HR|
|  3|  Varsha|     F|  3000|Payroll|
+---+--------+------+------+-------+
```

[ ]:

0.0.28 groupBy()

- Similar to SQL GROUP BY clause, PySpark groupBy() function is used to collect the identical data into groups on DataFrame and perform count, sum, avg, min, max functions on the grouped data

[ ]:
```
data= [
    (1, 'Ashutosh', 'M', 5000, 'IT'),
     (2, 'Keshav', 'M', 6000, 'IT'),
      (3, 'Varsha', 'F',2500, 'Payroll'),
       (4, 'Bhushan', 'M',4000, 'HR'),
        (5, 'Jiya', 'F', 2000, 'HR'),
         (6, 'Akash', 'M', 2000, 'Payroll'),
          (7, 'ayesha', 'F',3000, 'IT')
          ]

schema = ['id', 'name', 'gender', 'salary', 'dep']

df = spark.createDataFrame(data, schema)

df.groupBy('dep').count().show()

df.groupBy('dep', 'gender').count().show()

df.groupBy('dep').min('salary').show()

df.groupBy('dep').max('salary').show()

df.groupBy('dep').avg('salary').show()
```

```
+-------+-----+
|    dep|count|
+-------+-----+
|Payroll|    2|
|     IT|    3|
|     HR|    2|
+-------+-----+

+-------+------+-----+
|    dep|gender|count|
+-------+------+-----+
```

```
|Payroll|     F|    1|
|     IT|     M|    2|
|Payroll|     M|    1|
|     HR|     M|    1|
|     HR|     F|    1|
|     IT|     F|    1|
+-------+------+-----+

+-------+-----------+
|    dep|min(salary)|
+-------+-----------+
|Payroll|       2000|
|     IT|       3000|
|     HR|       2000|
+-------+-----------+

+-------+-----------+
|    dep|max(salary)|
+-------+-----------+
|Payroll|       2500|
|     IT|       6000|
|     HR|       4000|
+-------+-----------+

+-------+-----------------+
|    dep|      avg(salary)|
+-------+-----------------+
|Payroll|           2250.0|
|     IT|4666.666666666667|
|     HR|           3000.0|
+-------+-----------------+
```

0.0.29 groupBy().agg()

- PySpark Groupby agg() is used to calculate more than one aggregate (multiple aggregates) at a time on grouped Data Frame

[ ]:
```
from pyspark.sql import functions as F

data= [(1, 'Ashutosh', 'M', 5000, 'IT'), (2, 'Keshav', 'M',6000, 'IT'),(3,␣
 ↪'Varsha', 'F',2500, 'Payroll'), (4, 'Bhushan', 'M', 4000, 'HR'), (5, 'Jiya',␣
 ↪'F', 2000, 'HR'), (6, 'Akash', 'M', 2000, 'Payroll'),

      (7, 'ayesha', 'F',3000, 'IT')]

schema = ['id', 'name', 'gender', 'salary', 'dep']
```

```
df = spark.createDataFrame(data, schema)

df.groupBy('dep').count().show()



df.groupBy('dep').agg(F.count('*').alias('countOfEmps'), F.min('salary').alias␣
 ↪('minSal'), F.max ('salary').alias ('maxSal')).show()
```

```
+-------+-----+
|    dep|count|
+-------+-----+
|Payroll|    2|
|     IT|    3|
|     HR|    2|
+-------+-----+

+-------+-----------+------+------+
|    dep|countOfEmps|minSal|maxSal|
+-------+-----------+------+------+
|Payroll|          2|  2000|  2500|
|     IT|          3|  3000|  6000|
|     HR|          2|  2000|  4000|
+-------+-----------+------+------+
```

0.0.30 unionByName()

- unionByName() lets you to merge/union two DataFrames with a different number of columns (different schema) by passing allowMissingColumns with the value true

[ ]:
```
datal = [(1, 'Ashutosh', 'male')]
schema1 = ['id', 'name', 'gender']

data2 = [(1, 'Keshav', 2000)]
schema2 = ['id', 'name', 'salary']

df1 = spark.createDataFrame(datal, schema1)
df2 = spark.createDataFrame (data2, schema2)

df1.union(df2).show()

df1.unionByName(allowMissingColumns=True, other=df2).show()
```

```
+---+--------+------+
| id|    name|gender|
+---+--------+------+
|  1|Ashutosh|  male|
```

```
|  1|  Keshav|  2000|
+---+--------+------+

+---+--------+------+------+
| id|    name|gender|salary|
+---+--------+------+------+
|  1|Ashutosh|  male|  NULL|
|  1|  Keshav|  NULL|  2000|
+---+--------+------+------+
```

0.0.31 select()

- select() function is used to select single, multiple, column by index, all columns from the list and the nested columns from a DataFrame

[ ]:
```
data = [(1, 'Ashutosh', 'male', 2000), (2, 'Keshav', 'male', 3000), (3,␣
 ↪'Varsha', 'female', 2500)]

schema = ['id', 'name', 'gender', 'salary']

df = spark.createDataFrame (data, schema)

#select single or multiple columns df.select('id', 'name').show()
df.select(df.id,df.name).show()
df.select(df['id'],df['name']).show()

#using col() function
from pyspark.sql. functions import col
df.select(col('id'), col('name')).show()
df.select(['id', 'name']).show()
```

```
+---+--------+
| id|    name|
+---+--------+
|  1|Ashutosh|
|  2|  Keshav|
|  3|  Varsha|
+---+--------+

+---+--------+
| id|    name|
+---+--------+
|  1|Ashutosh|
|  2|  Keshav|
|  3|  Varsha|
+---+--------+
```

```
+---+--------+
| id|    name|
+---+--------+
|  1|Ashutosh|
|  2|  Keshav|
|  3|  Varsha|
+---+--------+

+---+--------+
| id|    name|
+---+--------+
|  1|Ashutosh|
|  2|  Keshav|
|  3|  Varsha|
+---+--------+
```

0.0.32 join()

- join() is like SQL JOIN. We can combine columns from different DataFrames based on condition. It supports all basic join types such as INNER, LEFT,OUTER, RIGHT OUTER, LEFT ANTI, LEFT SEMI, CROSS, SELF
- leftsemi join similar to inner join but get columns only from left dataframe for matching rows.
- leftanti opposite to leftsemi, it gets not matching rows from left dataframe. Self join, joins data with same dataframe

[ ]:
```
datal = [(1, 'Ashutosh',2000,2), (2, 'Keshav',3000,1), (3, 'Bhushan', 1000,4)]
schema1 = ['id', 'name', 'salary', 'dep']
empDf = spark.createDataFrame(datal, schema1)

data2 = [(1, 'IT'), (2, 'HR'), (3, 'Payroll')]
schema2 = ['id', 'name']
depDf = spark.createDataFrame (data2, schema2)

empDf.show()
depDf.show()

empDf.join(depDf, empDf.dep==depDf.id, 'inner').show()
empDf.join(depDf, empDf.dep==depDf.id, 'left').show()
empDf.join(depDf, empDf.dep==depDf.id, 'right').show()
empDf.join(depDf, empDf.dep==depDf.id, 'full').show()
```

```
+---+--------+------+---+
| id|    name|salary|dep|
+---+--------+------+---+
|  1|Ashutosh|  2000|  2|
|  2|  Keshav|  3000|  1|
```

```
|  3| Bhushan|  1000|  4|
+---+--------+------+---+

+---+-------+
| id|   name|
+---+-------+
|  1|     IT|
|  2|     HR|
|  3|Payroll|
+---+-------+

+---+--------+------+---+---+----+
| id|    name|salary|dep| id|name|
+---+--------+------+---+---+----+
|  2|  Keshav|  3000|  1|  1|  IT|
|  1|Ashutosh|  2000|  2|  2|  HR|
+---+--------+------+---+---+----+

+---+--------+------+---+----+----+
| id|    name|salary|dep|  id|name|
+---+--------+------+---+----+----+
|  1|Ashutosh|  2000|  2|   2|  HR|
|  2|  Keshav|  3000|  1|   1|  IT|
|  3| Bhushan|  1000|  4|NULL|NULL|
+---+--------+------+---+----+----+

+----+--------+------+----+---+-------+
|  id|    name|salary| dep| id|   name|
+----+--------+------+----+---+-------+
|   2|  Keshav|  3000|   1|  1|     IT|
|NULL|    NULL|  NULL|NULL|  3|Payroll|
|   1|Ashutosh|  2000|   2|  2|     HR|
+----+--------+------+----+---+-------+

+----+--------+------+----+----+-------+
|  id|    name|salary| dep|  id|   name|
+----+--------+------+----+----+-------+
|   2|  Keshav|  3000|   1|   1|     IT|
|   1|Ashutosh|  2000|   2|   2|     HR|
|NULL|    NULL|  NULL|NULL|   3|Payroll|
|   3| Bhushan|  1000|   4|NULL|   NULL|
+----+--------+------+----+----+-------+
```

0.0.33 pivot() function

- It's used to rotate data in one column into multiple columns.

- It is an aggregation where one of the grouping column values will be converted in individual columns.

[ ]:
```
data = [(1, 'Ashutosh', 'male', 'IT'),
        (2, 'Keshav', 'male', 'IT'),
        (3, 'Varsha', 'female', 'HR'),
        (4, 'Jiya', 'female', 'IT'),
        (5, 'shakti', 'female', 'IT'),
        (6, 'Bhushan', 'male', 'HR'),
        (7, 'akash', 'male', 'HR'),
        (8, 'ayesha', 'female', 'IT')
]

schema = ['id', 'name', 'gender', 'dep']

df = spark.createDataFrame (data, schema)

df.show()

df.groupBy('dep').pivot('gender').count().show()
```

```
+---+--------+------+---+
| id|    name|gender|dep|
+---+--------+------+---+
|  1|Ashutosh|  male| IT|
|  2|  Keshav|  male| IT|
|  3|  Varsha|female| HR|
|  4|    Jiya|female| IT|
|  5|  shakti|female| IT|
|  6| Bhushan|  male| HR|
|  7|   akash|  male| HR|
|  8|  ayesha|female| IT|
+---+--------+------+---+

+---+------+----+
|dep|female|male|
+---+------+----+
| HR|     1|   2|
| IT|     3|   2|
+---+------+----+
```

0.0.34 unpivot Dataframe

- Unpivot is rotating columns into rows. PySpark SQL doesn't have unpivot function hence will use the stack() function.

[ ]:
```
from pyspark.sql import functions as F

data = [('IT', 8, 5),('Payroll', 3, 2),('HR', 2, 4)]

schema = ['dep', 'male', 'female']

df = spark.createDataFrame (data, schema)
df.show()


df.select('dep', F.expr("stack (2, 'Male', male, 'Female', female) as (gender,␣
↪count)")).show()
```

```
+-------+----+------+
|    dep|male|female|
+-------+----+------+
|     IT|   8|     5|
|Payroll|   3|     2|
|     HR|   2|     4|
+-------+----+------+

+-------+------+-----+
|    dep|gender|count|
+-------+------+-----+
|     IT|  Male|    8|
|     IT|Female|    5|
|Payroll|  Male|    3|
|Payroll|Female|    2|
|     HR|  Male|    2|
|     HR|Female|    4|
+-------+------+-----+
```

0.0.35 fill() & fillna()

- fillna() or DataFrameNaFunctions.fill() is used to replace NULL/None values on all or selected multiple DataFrame columns with either zero(0), empty string, space, or any constant literal values.

[ ]:
```
data = [(1, 'Ashutosh', 'male', 1000, None), (2, 'Varsha', 'Female', 2000,␣
↪'IT'), (3, 'Keshav', None, 1000, 'HR')]

schema = ['id', 'name', 'gender', 'salary', 'dep']

df = spark.createDataFrame (data, schema)
df.show()

df.na.fill('No_value', ['gender']).show()
```

```
df.fillna('No_value', ['dep']).show()
```

```
+---+--------+------+------+----+
| id|    name|gender|salary| dep|
+---+--------+------+------+----+
|  1|Ashutosh|  male|  1000|NULL|
|  2|  Varsha|Female|  2000|  IT|
|  3|  Keshav|  NULL|  1000|  HR|
+---+--------+------+------+----+

+---+--------+--------+------+----+
| id|    name|  gender|salary| dep|
+---+--------+--------+------+----+
|  1|Ashutosh|    male|  1000|NULL|
|  2|  Varsha|  Female|  2000|  IT|
|  3|  Keshav|No_value|  1000|  HR|
+---+--------+--------+------+----+

+---+--------+------+------+--------+
| id|    name|gender|salary|     dep|
+---+--------+------+------+--------+
|  1|Ashutosh|  male|  1000|No_value|
|  2|  Varsha|Female|  2000|      IT|
|  3|  Keshav|  NULL|  1000|      HR|
+---+--------+------+------+--------+
```

0.0.36 Sample()

- To get the random sampling subset from the large dataset
- Use fraction to indicate what percentage of data to return and seed value to make sure every time to get same random sample.

[ ]:
```
# Assume df as big data

df = spark.range(start=1, end=101)

# fraction prameter takes a numeric value
# fraction = 0.1 --> 10% of data
df1 = df.sample(fraction=0.1, seed=123)

df2 = df.sample(fraction=0.1, seed=123)

display(df1)
display(df2)
```

0.0.37 collect()

- collect() retrieves all elements in a DataFrame as an Array of Row type to the driver node.
- collect() is an action hence it does not return a DataFrame instead, it returns data in an Array to the driver. Once the data is in an array, you can use python for loop to process it further.
- collect() use it with small DataFrames. With big DataFrames it may result in out of memory error as its return entier data to single node(driver)

[ ]:
```
data = [(1, 'Ashutosh', 'male', 1000, None), (2, 'Varsha', 'Female', 2000,␣
 ↪'IT'), (3, 'Bhushan', None, 1000, 'HR')]
schema = ['id', 'name', 'gender', 'salary', 'dep']

df = spark.createDataFrame (data, schema)
df.show()

dataRows = df.collect()
print(dataRows)

print(dataRows[0])

print(dataRows[0][0])
```

```
+---+--------+------+------+----+
| id|    name|gender|salary| dep|
+---+--------+------+------+----+
|  1|Ashutosh|  male|  1000|NULL|
|  2|  Varsha|Female|  2000|  IT|
|  3| Bhushan|  NULL|  1000|  HR|
+---+--------+------+------+----+

[Row(id=1, name='Ashutosh', gender='male', salary=1000, dep=None), Row(id=2,
name='Varsha', gender='Female', salary=2000, dep='IT'), Row(id=3,
name='Bhushan', gender=None, salary=1000, dep='HR')]
Row(id=1, name='Ashutosh', gender='male', salary=1000, dep=None)
1
```

0.0.38 DataFrame.transform()

- It's is used to chain the custom transformations and this function returns the new DataFrame after applying the specified transformations.

[ ]:
```
from pyspark.sql import functions as F

data = [(1, 'Ashutosh', 2000), (2, 'Bhushan', 3000)]
schema = ['id', 'name', 'salary']

df = spark.createDataFrame (data, schema)
```

```
df.show()


def convertToUpper (df):
  return df.withColumn('name', F.upper (df.name))

def doubleTheSalary (df):
  return df.withColumn('salary', df.salary * 2)

df1 = df.transform(convertToUpper).transform(doubleTheSalary)
df1.show()
```

```
+---+--------+------+
| id|    name|salary|
+---+--------+------+
|  1|Ashutosh|  2000|
|  2| Bhushan|  3000|
+---+--------+------+

+---+--------+------+
| id|    name|salary|
+---+--------+------+
|  1|ASHUTOSH|  4000|
|  2| BHUSHAN|  6000|
+---+--------+------+
```

0.0.39 pyspark.sql.functions.transform()

It's is used to apply the transformation on a column of type Array. This function applies the specified transformation on every element of the array and returns an object of ArrayType.

[ ]:
```
from pyspark.sql import functions as F

data = [(1, 'Ashutosh', ['azure', 'dotnt']), (2, 'Keshav', ['aws', 'java'])]
schema = ['id', 'name','skills']

df = spark.createDataFrame(data, schema)
df.show()
df.printSchema()

df.select('id', 'name', F.transform('skills', lambda x: F.upper (x)).
 ↪alias('skills')).show()

def convertToUpper1(x):
  return F.upper(x)

df.select(F.transform('skills', convertToUpper1)).show()
```

```
+---+--------+--------------+
| id|    name|        skills|
+---+--------+--------------+
|  1|Ashutosh|[azure, dotnt]|
|  2|  Keshav|   [aws, java]|
+---+--------+--------------+

root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- skills: array (nullable = true)
 |    |-- element: string (containsNull = true)

+---+--------+--------------+
| id|    name|        skills|
+---+--------+--------------+
|  1|Ashutosh|[AZURE, DOTNT]|
|  2|  Keshav|   [AWS, JAVA]|
+---+--------+--------------+

+--------------------------------------------------------------------------------
-------+
|transform(skills, lambdafunction(upper(namedlambdavariable()),
namedlambdavariable()))|
+--------------------------------------------------------------------------------
-------+
|                                                                          [AZURE,
DOTNT]|
|
[AWS, JAVA]|
+--------------------------------------------------------------------------------
-------+
```

0.0.40 createOrReplaceTempView()

- Advantage of Spark, you can work with SQL along with DataFrames. That means, if you are comfortable with SQL, you can create temporary view on Dataframe by using createOrReplace TempView() and use SQL to select and manipulate data.
- Temp Views are session scoped and cannot be shared between the sessions.

[ ]:
```
data = [(1, 'Ashutosh', 2000), (2, 'Bhushan',3000)]

schema = ['id', 'name', 'salary']

df = spark.createDataFrame(data, schema)
```

```
# will create a temp table called employees
df.createOrReplaceTempView('employees')

df1 = spark.sql('SELECT * FROM employees')
df1.show()
```

```
+---+--------+------+
| id|    name|salary|
+---+--------+------+
|  1|Ashutosh|  2000|
|  2| Bhushan|  3000|
+---+--------+------+
```

[ ]:
```
## Note: Instade of using spark.sql() function we can normally use sql queries␣
↪by using magic function


data = [(1, 'Ashutosh', 2000), (2, 'Bhushan',3000)]

schema = ['id', 'name', 'salary']

df = spark.createDataFrame(data, schema)

# will create a temp table called employees
df.createOrReplaceTempView('employees')

df1 = spark.sql('SELECT * FROM employees')
df1.show()
```

```
+---+--------+------+
| id|    name|salary|
+---+--------+------+
|  1|Ashutosh|  2000|
|  2| Bhushan|  3000|
+---+--------+------+
```

0.0.41 UDF(user defined function)

- These are similar to functions in SQL. We define some logic in functions and store them in Database and use them in queries.
- Similar to that we can write our own custom logic in python function and register it with PySpark using udf() function

[ ]:
```
# Way 1
```

```
from pyspark.sql import functions as F
from pyspark.sql import types as T


data = [(1, 'Ashutosh', 2000, 500), (2, 'Keshav', 4000, 1000)]

schema = ['id', 'name', 'salary', 'bonus']

df = spark.createDataFrame (data, schema)
df.show()

def totalPay (s,b):
  return s+b

# registering a udf
TotalPay = F.udf(lambda x,y: totalPay(x,y), T.IntegerType()) # --> F.udf(lambda␣
 ↪{parameter_1}, {parameter_2}: udf_name(parameter_1,parameter_2),␣
 ↪{udf_return_Type()})

df.select('*', TotalPay (F.col('salary'), F.col('bonus')).alias('totalPay')).
 ↪show()

# calling udf
df.withColumn('totalPay', TotalPay(df. salary, df.bonus)).show()
```

```
+---+--------+------+-----+
| id|    name|salary|bonus|
+---+--------+------+-----+
|  1|Ashutosh|  2000|  500|
|  2|  Keshav|  4000| 1000|
+---+--------+------+-----+

+---+--------+------+-----+--------+
| id|    name|salary|bonus|totalPay|
+---+--------+------+-----+--------+
|  1|Ashutosh|  2000|  500|    2500|
|  2|  Keshav|  4000| 1000|    5000|
+---+--------+------+-----+--------+

+---+--------+------+-----+--------+
| id|    name|salary|bonus|totalPay|
+---+--------+------+-----+--------+
|  1|Ashutosh|  2000|  500|    2500|
|  2|  Keshav|  4000| 1000|    5000|
+---+--------+------+-----+--------+
```

[ ]:
```
# way 2

from pyspark.sql import functions as F
from pyspark.sql.functions import udf
from pyspark.sql import types as T

data =[(1, 'Ashutosh', 2000, 500), (2, 'Varsha', 4000, 1000)]
schema = ['id', 'name', 'salary', 'bonus']

df = spark.createDataFrame (data, schema)
df.show()

@udf (returnType = T.IntegerType())
def totalPay(s,b):
  return s+b

# calling udf
df.select('*', totalPay(F.col('salary'),F.col('bonus')).alias('totalPay')).
 ↪show()

df.withColumn('totalPay', totalPay(df.salary, df.bonus)).show()
```

```
+---+--------+------+-----+
| id|    name|salary|bonus|
+---+--------+------+-----+
|  1|Ashutosh|  2000|  500|
|  2|  Varsha|  4000| 1000|
+---+--------+------+-----+

+---+--------+------+-----+--------+
| id|    name|salary|bonus|totalPay|
+---+--------+------+-----+--------+
|  1|Ashutosh|  2000|  500|    2500|
|  2|  Varsha|  4000| 1000|    5000|
+---+--------+------+-----+--------+

+---+--------+------+-----+--------+
| id|    name|salary|bonus|totalPay|
+---+--------+------+-----+--------+
|  1|Ashutosh|  2000|  500|    2500|
|  2|  Varsha|  4000| 1000|    5000|
+---+--------+------+-----+--------+
```

[ ]:
```
# way 3 using udf with createOrReplaceTempView()

from pyspark.sql import functions as F
```

```
from pyspark.sql.functions import udf
from pyspark.sql import types as T


data = [(1, 'Ashutosh', 2000, 500), (2, 'Vasha',4000, 1000)]

schema = ['id', 'name', 'salary', 'bonus']


df = spark.createDataFrame(data, schema)

# udf & createOrReplaceTempView()

# step 1: creating temporary view on Dataframe
df.createOrReplaceTempView('emps')

# step 2: create python function
def totalPay (s,b):
  return s+b

# step 3: to register udf function so that we can use in SQL queries
spark.udf.register(name='TotalPaySQL', f=totalPay, returnType=T.IntegerType())

# step 4: using udf in SQL querie
df = spark.sql('SELECT *, TotalPaySQL(salary, bonus) FROM emps')

df.show()
```

```
+---+--------+------+-----+-------------------------+
| id|    name|salary|bonus|TotalPaySQL(salary, bonus)|
+---+--------+------+-----+-------------------------+
|  1|Ashutosh|  2000|  500|                     2500|
|  2|   Vasha|  4000| 1000|                     5000|
+---+--------+------+-----+-------------------------+
```

0.0.42 Convert RDD(Resilient Distributed Dataset) to Dataframe

- Its collection of objects similar to list in Python. Its Immutable and In memory processing.
- By using parallelize() function of SparkContext you can create an RDD.

[ ]:
```
data = [(1, 'Ashutosh'), (2, 'Varsha')]

# creating rdd
rdd = spark.sparkContext.parallelize(data)
print(rdd.collect())
```

```
#converting rdd to dataframe

# way 1
df = rdd.toDF(schema=['id', 'name'])
df.show()

# way 2
df1 = spark.createDataFrame(rdd, ['id', 'name'])
df1.show()
```

```
[(1, 'Ashutosh'), (2, 'Varsha')]
+---+--------+
| id|    name|
+---+--------+
|  1|Ashutosh|
|  2|  Varsha|
+---+--------+

+---+--------+
| id|    name|
+---+--------+
|  1|Ashutosh|
|  2|  Varsha|
+---+--------+
```

0.0.43 map() transformation

- ts RDD transformation used to apply function(lambda) on every element of RDD and returns new RDD.
- Dataframe doesn't have map() transformation to use with Dataframe you need to generate RDD first.

[ ]:
```
data = [('Ashu', 'kumbhare'), ('Varsha', 'Kumbhare')]

rdd = spark.sparkContext.parallelize(data)

rdd1 = rdd.map(lambda x: x + (x[0]+' '+x[1],))
print(rdd1.collect())
```

```
[('Ashu', 'kumbhare', 'Ashu kumbhare'), ('Varsha', 'Kumbhare', 'Varsha
Kumbhare')]
```

0.0.44 flatMap() transformation

- flatMap() is a transformation operation that flattens the RDD (array/map DataFrame columns) after applying the function on every element and returns a new PySpark RDD
- Its not available in dataframes. Explode() functions can be used in dataframes to flatten

arrays.

[ ]:
```
data =['Ashu kumbhare', 'Varsha Kumbhare']

rdd = spark.sparkContext.parallelize(data)
print(rdd.collect())

for item in rdd.collect():
  print(item)

rdd1=rdd.flatMap(lambda x: x.split(' '))

for item in rdd1.collect():
  print(item)
```

```
['Ashu kumbhare', 'Varsha Kumbhare']
Ashu kumbhare
Varsha Kumbhare
Ashu
kumbhare
Varsha
Kumbhare
```

0.0.45 partitionBy function

- Its used to partition large Dataset into smaller files based on one or multiple columns

[ ]:
```
data = [(1, 'Ashutosh', 'male', 'IT'), (2, 'keshav', 'male', 'HR'), (3,␣
 ↪'varsha', 'female', 'IT')]

schema = ['id', 'name', 'gender', 'dep']

df = spark.createDataFrame(data, schema)

df.write.parquet ('path', mode= 'overwrite', partitionBy= 'dep')
```

0.0.46 from_json() function

- • It's used to convert json string in to MapType or StructType. In this video we discuss about converting it to MapType.

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql import types as T

data = [('Ashutosh', '{"hair": "black","eye": "brown"}')]

schema = ['name', 'props']
```

```
df = spark.createDataFrame (data, schema)

df.show(truncate=False)
df.printSchema()



#propsMap column with MapType gets generates from json sting
df1 = df.withColumn('propsMap', F.from_json(df.props, T.MapType(T.StringType(),␣
 ↪T.StringType())))

df1.show(truncate=False)
df1.printSchema()
display (df1)

#accessing 'eye' key from MapType column 'propsMap'
df2= df1.withColumn('eye', df1.propsMap.eye)
df2.show(truncate=False)
```

```
+--------+--------------------------------+
|name    |props                           |
+--------+--------------------------------+
|Ashutosh|{"hair": "black","eye": "brown"}|
+--------+--------------------------------+

root
 |-- name: string (nullable = true)
 |-- props: string (nullable = true)

+--------+--------------------------------+-----------------------------+
|name    |props                           |propsMap                     |
+--------+--------------------------------+-----------------------------+
|Ashutosh|{"hair": "black","eye": "brown"}|{hair -> black, eye -> brown}|
+--------+--------------------------------+-----------------------------+

root
 |-- name: string (nullable = true)
 |-- props: string (nullable = true)
 |-- propsMap: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)


DataFrame[name: string, props: string, propsMap: map<string,string>]

+--------+--------------------------------+-----------------------------+-----+
|name    |props                           |propsMap                     |eye  |
+--------+--------------------------------+-----------------------------+-----+
```

```
|Ashutosh|{"hair": "black","eye": "brown"}|{hair -> black, eye -> brown}|brown|
+--------+-------------------------------+-----------------------------+-----+
```

[ ]:
```
# It's used to convert json string in to MapType or StructType. In this video we discuss about converting it to MapType.

from pyspark.sql import functions as F
from pyspark.sql import types as T

data = [('Ashutosh', '{"hair": "black","eye": "brown"}')]

schema = ['name', 'props']

df = spark.createDataFrame (data, schema)

df.show(truncate=False)
df.printSchema()



structSchema = T.StructType([T.StructField('hair', T.StringType()), T.StructField('eye', T.StringType())])

df1 = df.withColumn('propsMap', F.from_json(df.props, structSchema))

df1.show(truncate=False)
df1.printSchema()

#accessing 'eye' key from MapType column 'propsMap'
df2= df1.withColumn('eye', df1.propsMap.eye)
df2.show(truncate=False)
```

```
+--------+-------------------------------+
|name    |props                          |
+--------+-------------------------------+
|Ashutosh|{"hair": "black","eye": "brown"}|
+--------+-------------------------------+

root
 |-- name: string (nullable = true)
 |-- props: string (nullable = true)

+--------+-------------------------------+--------------+
|name    |props                          |propsMap      |
+--------+-------------------------------+--------------+
|Ashutosh|{"hair": "black","eye": "brown"}|{black, brown}|
+--------+-------------------------------+--------------+
```

```
root
 |-- name: string (nullable = true)
 |-- props: string (nullable = true)
 |-- propsMap: struct (nullable = true)
 |    |-- hair: string (nullable = true)
 |    |-- eye: string (nullable = true)

+--------+--------------------------------+--------------+-----+
|name    |props                           |propsMap      |eye  |
+--------+--------------------------------+--------------+-----+
|Ashutosh|{"hair": "black","eye": "brown"}|{black, brown}|brown|
+--------+--------------------------------+--------------+-----+
```

0.0.47 to_json() function

- to_json() is used to convert DataFrame column MapType or Struct Type to JSON string.

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql import types as T

data = [('Ashutosh', {'hair': 'black', 'eye': 'brown'})]

schema = ['name', 'properties']

df = spark.createDataFrame (data, schema)
df.show(truncate=False)
df.printSchema()

#here 'prop' column will get generate as json string
df1 = df.withColumn('prop', F.to_json(df.properties))

df1.show(truncate=False)
df1.printSchema()
```

```
+--------+-----------------------------+
|name    |properties                   |
+--------+-----------------------------+
|Ashutosh|{eye -> brown, hair -> black}|
+--------+-----------------------------+

root
 |-- name: string (nullable = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
```

```
+--------+-----------------------------+-----------------------------+
|name    |properties                   |prop                         |
+--------+-----------------------------+-----------------------------+
|Ashutosh|{eye -> brown, hair -> black}|{"eye":"brown","hair":"black"}|
+--------+-----------------------------+-----------------------------+

root
 |-- name: string (nullable = true)
 |-- properties: map (nullable = true)
 |    |-- key: string
 |    |-- value: string (valueContainsNull = true)
 |-- prop: string (nullable = true)
```

0.0.48 json_tuple() function

- json_tuple() function is used to query or extract elements from json string column and create as new columns.

[ ]:
```
from pyspark.sql import functions as F

data = [('Ashutosh', '{"hair":"black", "eye": "brown", "skin": "brown"}'),␣
 ↪('Bhushan', '{"hair":"black", "eye":"blue", "skin": "white"}')]
schema = ['name', 'props']

df = spark.createDataFrame (data, schema)
df.show(truncate=False)
df.printSchema()

df2 = df.select(df.name, F.json_tuple (df.props, 'hair', 'skin').
 ↪alias('hair','skin'))
df2.show()
```

```
+--------+-----------------------------------------------------+
|name    |props                                                |
+--------+-----------------------------------------------------+
|Ashutosh|{"hair":"black", "eye": "brown", "skin": "brown"}|
|Bhushan |{"hair":"black", "eye":"blue", "skin": "white"}  |
+--------+-----------------------------------------------------+

root
 |-- name: string (nullable = true)
 |-- props: string (nullable = true)

+--------+-----+-----+
|    name| hair| skin|
+--------+-----+-----+
|Ashutosh|black|brown|
```

```
| Bhushan|black|white|
+--------+-----+-----+
```

0.0.49 get_ json_object() function

- It's used to extract the JSON string based on path from the JSON column.

```
[ ]: from pyspark.sql import functions as F


data = [('Ashutosh', '{"address":{"city": "hyd", "state": "telengana"},"gender":
  ↪ "male"}'), ('Keshav', '{"address":{"city": "banglore", "state":
  ↪"karnataka"},"eye":"blue"}')]

schema = ['name', 'props']

df = spark.createDataFrame(data, schema)
df.show(truncate=False)
df.printSchema()


df1 = df.select('name', F.get_json_object('props', '$.address.city').
  ↪alias('city'),
                        F.get_json_object('props', '$.address.state').
  ↪alias('state'))

df1.show(truncate=False)
df1.printSchema()
```

```
+--------+-------------------------------------------------------------------+
|name    |props                                                              |
+--------+-------------------------------------------------------------------+
|Ashutosh|{"address":{"city": "hyd", "state": "telengana"},"gender": "male"}|
|Keshav  |{"address":{"city": "banglore", "state":"karnataka"},"eye":"blue"}|
+--------+-------------------------------------------------------------------+

root
 |-- name: string (nullable = true)
 |-- props: string (nullable = true)

+--------+--------+---------+
|name    |city    |state    |
+--------+--------+---------+
|Ashutosh|hyd     |telengana|
|Keshav  |banglore|karnataka|
+--------+--------+---------+
```

```
root
 |-- name: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
```

0.0.50 Date functions

- DateType default fomart is yyyy-MM-dd
- current_date() get the current system date. By default, the data will be returned in yyyy-dd-mm format.
- date_format() to parses the date and converts from yyyy-MM-dd to specified format.
- to_date() converts date string in to datetype. We need to specify format of date in the string in the function.

[ ]:
```
from pyspark.sql import functions as F

df = spark.range(2)

#datetype default format yyyy-MM-dd I
df1= df.withColumn('currentDate', F.current_date())
df1.show()
df1.printSchema()


df2= df1.withColumn('currentDate', F.date_format (df1.currentDate, 'yyyy.MM.
 ↪dd') )
df2.show()
df2.printSchema()


df3= df2.withColumn('currentDate', F.to_date (df2.currentDate, 'yyyy.MM.dd'))
df3.show()
df3.printSchema()
```

```
+---+-----------+
| id|currentDate|
+---+-----------+
|  0| 2024-02-14|
|  1| 2024-02-14|
+---+-----------+

root
 |-- id: long (nullable = false)
 |-- currentDate: date (nullable = false)

+---+-----------+
```

```
| id|currentDate|
+---+-----------+
|  0| 2024.02.14|
|  1| 2024.02.14|
+---+-----------+

root
 |-- id: long (nullable = false)
 |-- currentDate: string (nullable = false)

+---+-----------+
| id|currentDate|
+---+-----------+
|  0| 2024-02-14|
|  1| 2024-02-14|
+---+-----------+

root
 |-- id: long (nullable = false)
 |-- currentDate: date (nullable = true)
```

0.0.51 datediff(), months_between(), add_months(), date_add(), month(), year()

- DataType default format is yyyy-MM-dd

[ ]:
```
from pyspark.sql import functions as F


df = spark.createDataFrame([('2015-04-08', '2015-05-08')], ['d1', 'd2'])


# shows difference between 2 dates in days
df.withColumn('diff', F.datediff(df.d2, df.d1)).show() # -->  F.
 ↪datediff(start_date, end_date)

# shows difference between 2 dates in months
df.withColumn('months Between', F.months_between (df.d2, df.d1)).show() # --> ␣
 ↪F.months_between(start_date, end_date)

# will add months on specified date
df.withColumn ('addmonth', F.add_months (df.d2,4)).show() # --> F.add_months␣
 ↪(date, number_of_months_to_be_added)

# will subtract months on specified date
df.withColumn ('submonth', F.add_months (df.d2,-4)).show()

# will add days
```

```
df.withColumn ('addDate', F.date_add (df.d2,4)).show()

# will subtract days
df.withColumn('subDate', F.date_add(df.d2,-4)).show()

# will give year from the date
df.withColumn ('year', F.year(df.d2)).show()

# will give months from the date
df.withColumn ('month', F.month(df.d2)).show()
```

```
+----------+----------+----+
|        d1|        d2|diff|
+----------+----------+----+
|2015-04-08|2015-05-08|  30|
+----------+----------+----+

+----------+----------+--------------+
|        d1|        d2|months Between|
+----------+----------+--------------+
|2015-04-08|2015-05-08|           1.0|
+----------+----------+--------------+

+----------+----------+----------+
|        d1|        d2|  addmonth|
+----------+----------+----------+
|2015-04-08|2015-05-08|2015-09-08|
+----------+----------+----------+

+----------+----------+----------+
|        d1|        d2|  submonth|
+----------+----------+----------+
|2015-04-08|2015-05-08|2015-01-08|
+----------+----------+----------+

+----------+----------+----------+
|        d1|        d2|   addDate|
+----------+----------+----------+
|2015-04-08|2015-05-08|2015-05-12|
+----------+----------+----------+

+----------+----------+----------+
|        d1|        d2|   subDate|
+----------+----------+----------+
|2015-04-08|2015-05-08|2015-05-04|
+----------+----------+----------+
```

```
+----------+----------+----+
|        d1|        d2|year|
+----------+----------+----+
|2015-04-08|2015-05-08|2015|
+----------+----------+----+

+----------+----------+-----+
|        d1|        d2|month|
+----------+----------+-----+
|2015-04-08|2015-05-08|    5|
+----------+----------+-----+
```

0.0.52 Timestamp Functions

- TimestampType default fomart is yyyy-MM-dd HH:mm:ss.SS
- current_timestamp() get the current timestamp. By default, the data will be returne in default format.
- to_timestamp() converts timestamp string in to TimestampType. We need to specify format of timestamp in the string in the function.
- hour(), minute(), second() functions

[ ]:
```
from pyspark.sql import functions as F


df = spark.range(2)
df1 = df.withColumn('currentTimeStamp', F.current_timestamp())
df1.show(truncate=False)
df1.printSchema()

df2= df1.withColumn('timestampInSting', F.lit('12.25.2022 08.10.03'))
df2.show(truncate=False)
df2.printSchema()

df3 = df2.withColumn('timestampInSting', F.to_timestamp(df2.timestampInSting,␣
 ↪'MM.dd.yyyy HH.mm.ss'))
df3.show(truncate=False)
df3.printSchema()
df1.select('*', F.hour (df1.currentTimeStamp).alias('hour'), F.minute(df1.
 ↪currentTimeStamp).alias('minute'), F.second (df1.currentTimeStamp).
 ↪alias('seconds')).show(truncate=False)
```

```
+---+--------------------------+
|id |currentTimeStamp          |
+---+--------------------------+
|0  |2024-02-14 22:41:44.593129|
|1  |2024-02-14 22:41:44.593129|
```

```
+---+-------------------------+

root
 |-- id: long (nullable = false)
 |-- currentTimeStamp: timestamp (nullable = false)

+---+-------------------------+------------------+
|id |currentTimeStamp         |timestampInSting  |
+---+-------------------------+------------------+
|0  |2024-02-14 22:41:44.819816|12.25.2022 08.10.03|
|1  |2024-02-14 22:41:44.819816|12.25.2022 08.10.03|
+---+-------------------------+------------------+

root
 |-- id: long (nullable = false)
 |-- currentTimeStamp: timestamp (nullable = false)
 |-- timestampInSting: string (nullable = false)

+---+-------------------------+------------------+
|id |currentTimeStamp         |timestampInSting  |
+---+-------------------------+------------------+
|0  |2024-02-14 22:41:45.261703|2022-12-25 08:10:03|
|1  |2024-02-14 22:41:45.261703|2022-12-25 08:10:03|
+---+-------------------------+------------------+

root
 |-- id: long (nullable = false)
 |-- currentTimeStamp: timestamp (nullable = false)
 |-- timestampInSting: timestamp (nullable = true)

+---+-------------------------+----+------+-------+
|id |currentTimeStamp         |hour|minute|seconds|
+---+-------------------------+----+------+-------+
|0  |2024-02-14 22:41:45.526608|22  |41    |45     |
|1  |2024-02-14 22:41:45.526608|22  |41    |45     |
+---+-------------------------+----+------+-------+
```

0.0.53 Aggregate functions

- Aggregate functions operate on a group of rows and calculate a single return value for every group.
- approx_count_distinct() – returns the count of distinct items in a group of rows
- avg() – returns average of values in a group of rows
- collect_list() - returns all values from input column as list with duplicates * collect_set() - returns all values from input column as list without duplicates.

- count Distinct() - returns number of distinct elements in input column.
- count() - returns number of elements in a column.

[ ]:
```
from pyspark.sql import functions as F

simpleData = [("Ashutosh", "HR", 1500),("Keshav", "IT", 3000),("Bhushan", "HR",␣
 ↪1500)]
schema = ["employee_name", "department", "salary"]

df = spark.createDataFrame (data = simpleData, schema = schema)
df.show()

df.select(F.approx_count_distinct('salary')).show()

df.select(F.avg('salary')).show()

df.select(F.collect_list('salary')).show()

df.select(F.collect_set('salary')).show()

df.select(F.countDistinct('salary')).show()

df.select(F.count('salary')).show()
```

```
+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|     Ashutosh|        HR|  1500|
|       Keshav|        IT|  3000|
|      Bhushan|        HR|  1500|
+-------------+----------+------+

+----------------------------+
|approx_count_distinct(salary)|
+----------------------------+
|                           2|
+----------------------------+

+-----------+
|avg(salary)|
+-----------+
|     2000.0|
+-----------+

+--------------------+
|collect_list(salary)|
+--------------------+
```

```
|  [1500, 3000, 1500]|
+--------------------+

+-------------------+
|collect_set(salary)|
+-------------------+
|       [3000, 1500]|
+-------------------+

+----------------------+
|count(DISTINCT salary)|
+----------------------+
|                     2|
+----------------------+

+-------------+
|count(salary)|
+-------------+
|            3|
+-------------+
```

0.0.54 row_number(), rank(), dense_rank()

we need to partition the data using Window. partitionBy(), and for row number and rank function we need to additionally order by on partition data using orderBy clause.

- row_number() window function is used to give the sequential row number starting from 1 to the result of each window partition
- rank() window function is used to provide a rank to the result within alie window partition. This function leaves gaps in rank when there are ties.
- dense_rank() window function is used to get the result with rank of rows within a window partition without any gaps. This is similar to rank() function difference being rank function leaves gaps in rank when there are ties.

[ ]:
```
from pyspark.sql import functions as F
from pyspark.sql.window import Window

data = [
    ('Ashutosh', 'HR', 2000), ('Keshav', 'IT',3000), ('Bhushan', 'HR',1500),␣
 ↪('Jiya', 'payroll',3500),
    ('shakti', 'IT', 3000), ('Varsha', 'IT',4000), ('Ajay', 'payroll', 2000),␣
 ↪('Himanshu', 'IT', 2000),
    ('Akash', 'HR', 2000), ('Alex', 'IT', 2500)
    ]
schema = ['name', 'dep', 'salary']
```

```
df = spark.createDataFrame (data, schema)
df.show()

window = Window.partitionBy("dep").orderBy('salary')

df.withColumn('rowNumber', F.row_number().over (window)).show()

df.withColumn('rank', F.rank().over (window)).show()

df.withColumn('denserank', F.dense_rank().over (window)).show()
```

```
+--------+-------+------+
|    name|    dep|salary|
+--------+-------+------+
|Ashutosh|     HR|  2000|
|  Keshav|     IT|  3000|
| Bhushan|     HR|  1500|
|    Jiya|payroll|  3500|
|  shakti|     IT|  3000|
|  Varsha|     IT|  4000|
|    Ajay|payroll|  2000|
|Himanshu|     IT|  2000|
|   Akash|     HR|  2000|
|    Alex|     IT|  2500|
+--------+-------+------+

+--------+-------+------+---------+
|    name|    dep|salary|rowNumber|
+--------+-------+------+---------+
| Bhushan|     HR|  1500|        1|
|Ashutosh|     HR|  2000|        2|
|   Akash|     HR|  2000|        3|
|Himanshu|     IT|  2000|        1|
|    Alex|     IT|  2500|        2|
|  Keshav|     IT|  3000|        3|
|  shakti|     IT|  3000|        4|
|  Varsha|     IT|  4000|        5|
|    Ajay|payroll|  2000|        1|
|    Jiya|payroll|  3500|        2|
+--------+-------+------+---------+

+--------+-------+------+----+
|    name|    dep|salary|rank|
+--------+-------+------+----+
| Bhushan|     HR|  1500|   1|
|Ashutosh|     HR|  2000|   2|
|   Akash|     HR|  2000|   2|
```

```
|Himanshu|     IT|  2000|   1|
|    Alex|     IT|  2500|   2|
|  Keshav|     IT|  3000|   3|
|  shakti|     IT|  3000|   3|
|  Varsha|     IT|  4000|   5|
|    Ajay|payroll|  2000|   1|
|    Jiya|payroll|  3500|   2|
+--------+-------+------+----+

+--------+-------+------+---------+
|    name|    dep|salary|denserank|
+--------+-------+------+---------+
| Bhushan|     HR|  1500|        1|
|Ashutosh|     HR|  2000|        2|
|   Akash|     HR|  2000|        2|
|Himanshu|     IT|  2000|        1|
|    Alex|     IT|  2500|        2|
|  Keshav|     IT|  3000|        3|
|  shakti|     IT|  3000|        3|
|  Varsha|     IT|  4000|        4|
|    Ajay|payroll|  2000|        1|
|    Jiya|payroll|  3500|        2|
+--------+-------+------+---------+
```