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,'maheer','male',2000),(2,'wafa','male',3000),(3,'asi','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()
```

```
1  data = [(1,'maheer','male',2000)]
2  schema = ['id','name','gender','salary']
3      .
4  df = spark.createDataFrame(data,schema)
5  df.select('*').show()
```

Cancel Uploading command

df: pyspark.sql.dataframe.DataFrame = [id: long, nar

```
+---+------+
| id|  name|
+---+------+
|  1|maheer|
+---+------+
```

Select all Column from the table

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

```
data1 = [(1,'maheer',2000,2),(2,'wafa',3000,1),(3,'abcd',1000,4)]
schema1 = ['id','name','salary','dep']

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

empDf = spark.createDataFrame(data1,schema1)
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()
```

Join Comparison

Join Type	What You Get
Inner Join	Only matching rows from both DataFrames.
Left Join	All rows from the left, match from right.
Right Join	All rows from the right, match from left.
Full Join	All rows from both, with `NULL` for unmatched.

join()

- 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

```
data1 = [(1,'maheer',2000,2),(2,'wafa',3000,1),(3,'abcd',1000,4)]
schema1 = ['id','name','salary','dep']

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

empDf = spark.createDataFrame(data1,schema1)
depDf = spark.createDataFrame(data2,schema2)
empDf.show()
depDf.show()

empDf.join(empDf, empDf.id==empDf.id,'leftsemi').show()
empDf.join(depDf, empDf.dep==depDf.id,'leftanti').show()
```

```
data1 = [(1,'maheer',0),(2,'wafa',1),(3,'asi',2)]
schema1 = ['id','name','managerId']

df = spark.createDataFrame(data1,schema1)
from pyspark.sql.functions import col
df.alias('emp').join(df.alias('manager'), col('emp.managerId') == col('manager.id'),'left')\
    .select(col('emp.name').alias('empName'),col('manager.name').alias('managerName')).show()
```

pivot()

- 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,'maheer','male','IT'),\
        (2,'wafa','male','IT'),\
        (3,'asi','female','HR'),\
        (4,'annu','female','IT'),\
         (5,'shakti','female','IT'),\
         (6,'pradeep','male','HR'),\
         (7,'sarfaraj','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()
```

```
df.groupBy('dep').pivot('gender',['male','female']).count().show()
```

```
+---+--------+------+---+
| id|    name|gender|dep|
+---+--------+------+---+
|  1|  maheer|  male| IT|
|  2|    wafa|  male| IT|
|  3|     asi|female| HR|
|  4|    annu|female| IT|
|  5|  shakti|female| IT|
|  6| pradeep|  male| HR|
|  7|sarfaraj|  male| HR|
|  8|  ayesha|female| IT|
+---+--------+------+---+
```

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

Unpivot Dataframe

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

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

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

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

from pyspark.sql.functions import expr
df.select('dep', expr("stack(2, 'Male', male, 'Female', female) as (gender, count)")).show()
```

how many columns (male, female)

```
+-------+----+------+
|    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|
+-------+------+-----+
```

Pivot V/s Unpivot

Aspect	Pivot	Unpivot
Direction	Converts rows into columns.	Converts columns into rows.
Use Case	Used for summarizing or aggregating data.	Used for restructuring data to long format.
Output Format	Wider table (more columns).	Longer table (more rows).

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.

Note:- fill & fillna both are same

```
data = [(1,'Maheer','male',1000,None),(2,'Asi','Female',2000,'IT'),(3,'abcd',None,1000,'HR')]
schema = ['id','name','gender','salary','dep']

df = spark.createDataFrame(data,schema)
df.show()
df.na.fill('unknown',['gender']).show()
df.fillna('unknown',['dep']).show()
```

I/P

```
+---+------+------+------+----+
| id|  name|gender|salary| dep|
+---+------+------+------+----+
|  1|Maheer|  male|  1000|null|
|  2|   Asi|Female|  2000|  IT|
|  3|  abcd|  null|  1000|  HR|
+---+------+------+------+----+
```

O/P

```
+---+------+-------+------+-------+
| id|  name| gender|salary|    dep|
+---+------+-------+------+-------+
|  1|Maheer|   male|  1000|unknown|
|  2|   Asi| Female|  2000|     IT|
|  3|  abcd|unknown|  1000|     HR|
+---+------+-------+------+-------+
```

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.

```
df = spark.range(start=1, end=101)
df1 = df.sample(fraction=0.1, seed=123)
df2 = df.sample(fraction=0.1, seed=123)
display(df1)
display(df2)
```

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,'Maheer','male',1000,None),(2,'Asi','Female',2000,'IT'),(3,'abcd',None,1000,'HR')]
schema = ['id','name','gender','salary','dep']

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

dataRows = df.collect()
print(dataRows)
print(dataRows[0])
```

O/P

```
+---+------+------+
| id|  name|salary|
+---+------+------+
|  1|maheer|  2000|
|  2|  wafa|  3000|
```

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

```
[Row(id=1, name='maheer', salary=2000), Row(id=2, name='wafa', salary=3000)]
Row(id=1, name='maheer', salary=2000)
1
```

Dataframe.transform()

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

```
data = [(1,'maheer', 2000),(2,'wafa', 3000)]
schema = ['id','name','salary']

df = spark.createDataFrame(data,schema)

df.show()
from pyspark.sql.functions import upper
def convertToUpper(df):
    return df.withColumn('name', upper(df.name))

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

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

I/P

```
+---+------+------+
|  1|maheer|  2000|
|  2|  wafa|  3000|
+---+------+------+
```

O/P

```
+---+------+------+
| id|  name|salary|
+---+------+------+
|  1|MAHEER|  2000|
|  2|  WAFA|  3000|
+---+------+------+
```

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 createOrReplaceTempView() and use SQL to select and manipulate data.
- Temp Views are session scoped and cannot be shared between the sessions.

```
data = [(1,'maheer',2000),(2,'asi',3000)]
schema = ['id','name','salary']

df = spark.createDataFrame(data, schema)

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

* This will be available with in the Session.
* We Can't use this Tep view another Notebook

```
%sql
SELECT id, UPPER(name) FROM employees
```

createOrReplaceGlobalTempView()

- It's used to create temp views or tables globally, when can be accessed across the sessions with in Spark Application.
- To query these tables, we need append global_temp.<tablename>

```
# scala code to get spark session id
%scala
spark
```

```
# list tables from current sessions
spark.catalog.listTables(spark.catalog.currentDatabase())

#To view global temp tables
spark.catalog.listTables('global_temp')
```

```
data = [(1,'maheer',2000)]
schema = ['id','name','salary']
df = spark.createDataFrame(data,schema)
df.createOrReplaceGlobalTempView('empGlobal')
```

df1 = df.Select(id, Name(Global_table.EmpGlobal))

df.show()

V/S

Aspect	Temporary View	Global Temporary View
Scope	Limited to the current Spark session.	Shared across all Spark sessions in the application.
Lifetime	Exists only as long as the current session is active.	Exists as long as the Spark application is running.
Accessibility	Accessible only within the session that created it.	Accessible across multiple sessions or notebooks.
Usage	Use for session-specific operations.	Use for sharing data between sessions or users.
Creation Command	`createOrReplaceTempView`	`createOrReplaceGlobalTempView`
Storage Location	Stored in the session catalog.	Stored in the **global temporary database** (`global_temp`).
Querying	Can be queried directly by its name.	Must be queried using the `global_temp` prefix.

UDF

There are two ways to define a UDF 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

First Way

```
data =[(1,'maheer',2000, 500),(2,'wafa',4000, 1000)]

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

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

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

from pyspark.sql.functions import udf,col
from pyspark.sql.types import IntegerType

TotalPay = udf(lambda x,y: totalPay(x,y), IntegerType())

df.select('*',TotalPay(col('salary'),col('bonus')).alias('totalPay')).show()
df.withColumn('totPay', TotalPay(df.salary,df.bonus)).show()
```

We can define the custom function in Step - ①

① ② ③ Result

Second Way

```
1  @udf(returnType=IntegerType())
2  def totalPay(s,b):
3      return s+b
```

① ②

Command took 0.06 seconds -- by maheer3222@gmail.com at 12/26/202

Cmd 5

```
1  df.select('*', totalPay(df.salary,df.bonus)).show()
```

③

(3) Spark Jobs

o/p

```
+---+------+------+-----+----------------------+
| id|  name|salary|bonus|totalPay(salary, bonus)|
+---+------+------+-----+----------------------+
|  1|maheer|  2000|  500|                  2500|
|  2|  wafa|  4000| 1000|                  5000|
+---+------+------+-----+----------------------+
```

* There are two ways to Create RDD

First one ↓

RDD(Resilient Distributed Dataset)

- 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,'maheer'),(2,'wafa')]
rdd = spark.sparkContext.parallelize(data)
print(rdd.collect())
```


Convert RDD to Dataframe

Second ↑

```
df = rdd.toDF(['id','name'])
df.show()
```

```
df=spark.createDataFrame(rdd,['id','name'])
df.show()
```

Example: → RDD

Imagine you have a list of numbers: `[1, 2, 3, 4, 5]`. An RDD is like splitting that list into smaller chunks that can be processed on multiple computers at the same time. You can apply operations like "multiply each number by 2," and Spark will do it efficiently across those chunks.

So, RDDs help Spark handle large datasets in a smart way by spreading out the work, making things faster and more reliable.

map()

- Its 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 =[('maheer','shaik'),('wafa','shaik')]

rdd = spark.sparkContext.parallelize(data)

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

```
data =[('maheer','shaik'),('wafa','shaik')]
df = spark.createDataFrame(data,['fn','ln'])
rdd1 = df.rdd.map(lambda x: x + (x[0] + x[1],))
df1 = rdd1.toDF(['fn','ln','fullname'])
df1.show()
```

```
print(rdd1.collect())
```

```
def fullname(x):
    x = x + (x[0] + ' ' + x[1],)
    return x

data =[('maheer','shaik'),('wafa','shaik')]
df = spark.createDataFrame(data,['fn','ln'])
rdd1 = df.rdd.map(lambda x: fullname(x))
df1 = rdd1.toDF(['fn','ln','fullname'])
df1.show()
```

flatMap()

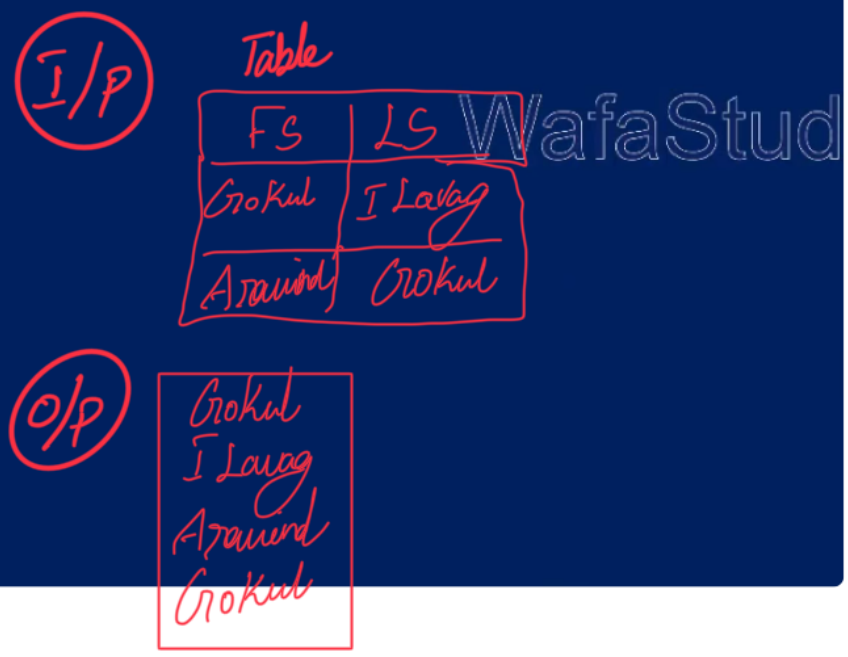
- flatMap() is a transformation operation that flattens the RDD (array/map DataFrame columns) after applying the function on every element and returns a new PySpark RDD
- Its not available in dataframes. Explode() functions can be used in dataframes to flatten arrays.

```
data =['maheer shaik','wafa shaik']

rdd = spark.sparkContext.parallelize(data)

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

rdd1=rdd.flatMap(lambda x: x.split(' '))
for item in rdd1.collect():
    print(item)
```



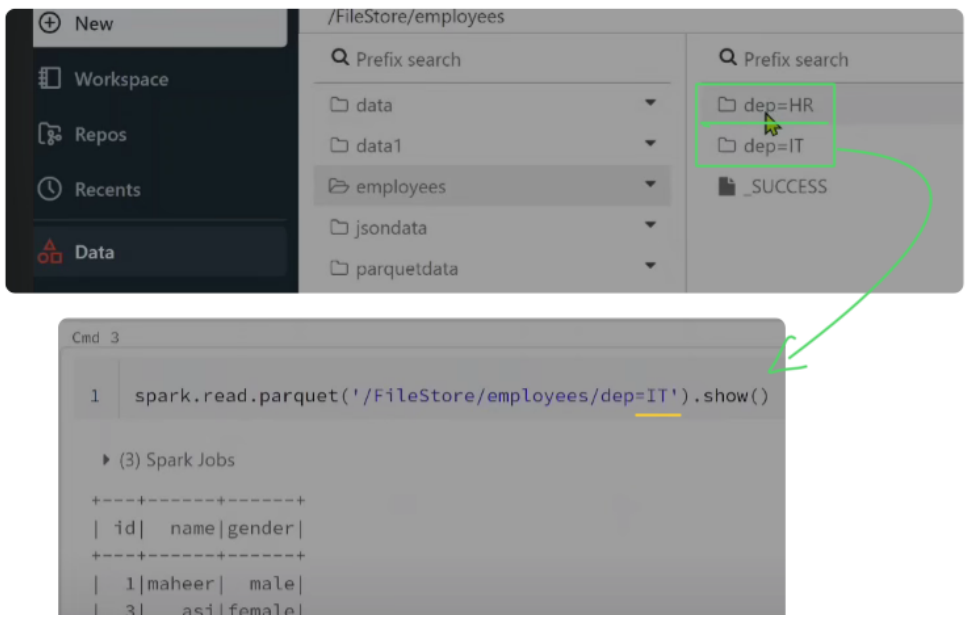
partitionBy()

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

```
data = [(1,'maheer','male','IT'),(2,'wafa','male','HR'),(3,'asi','female','IT')]
schema = ['id','name','gender','dep']

df = spark.createDataFrame(data,schema)
df.write.parquet(path='/FileStore/data/outputemps/',mode='overwrite',partitionBy='dep')
df.write.parquet(path='/FileStore/data/outputemps1/',mode='overwrite',partitionBy=['dep','gender'])
```


partition by dep →



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.

String type

```
from pyspark.sql.functions import current_date,date_format,lit,to_date

df = spark.range(1)
# datatype default format is yyyy-MM-dd
# gives current date in yyyy-MM-dd format
df.withColumn('todaysDate', current_date()).show()
# converts yyyy-MM-dd datetype to specified format
df.withColumn('newFormat', date_format(lit('2023-12-25'),'MM.dd.yyyy')).show()
# converts string values of date to DateType
df.withColumn('newDateCol', to_date(lit('25-12-2023'),'dd-MM-yyyy')).show()
```


Date Functions

- DateType default fomart is yyyy-MM-dd

```
from pyspark.sql.functions import datediff, months_between, add_months, date_add,year, month
df = spark.createDataFrame([('2015-04-08','2015-05-08')], ['d1', 'd2'])
df.withColumn('diff', datediff(df.d2, df.d1)).show()
df.withColumn('monthsBetween', months_between(df.d2, df.d1)).show()
df.withColumn('addmonth', add_months(df.d2,4)).show()
df.withColumn('submonth', add_months(df.d2,-4)).show()
df.withColumn('addDate', date_add(df.d2,4)).show()
df.withColumn('subDate', date_add(df.d2,-4)).show()
df.withColumn('year', year(df.d2)).show()
df.withColumn('month', month(df.d2)).show()
```


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 returned 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.functions import current_timestamp, to_timestamp, lit, hour, minute, second
df = spark.range(1)
df.show()
df1 = df.withColumn('timestamp', current_timestamp())
df1.show(truncate=False)
df1.printSchema()
df2 = df1.withColumn('toTimestamp', to_timestamp(lit('25.12.2022 06.10.13'), 'dd.MM.yyyy HH.mm.ss'))
df2.show(truncate=False)
df2.printSchema()
df2.select('id', hour(current_timestamp()).alias('hour'),\
          minute(current_timestamp()).alias('min'),\
          second(current_timestamp()).alias('second')).show()
```

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 returned 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.functions import current_timestamp, to_timestamp, lit, hour, minute, second
df = spark.range(1)
df.show()
df1 = df.withColumn('timestamp', current_timestamp())
df1.show(truncate=False)
df1.printSchema()
df2 = df1.withColumn('toTimestamp', to_timestamp(lit('25.12.2022 06.10.13'), 'dd.MM.yyyy HH.mm.ss'))
df2.show(truncate=False)
df2.printSchema()
df2.select('id', hour(current_timestamp()).alias('hour'),\
           minute(current_timestamp()).alias('min'),\
           second(current_timestamp()).alias('second')).show()
```

```
+---+-----------------------+-----------------------+
|id |currentTimeStamp       |timestampInSting       |
+---+-----------------------+-----------------------+
|0  |2023-02-10 14:13:25.795|2022-12-25 08:10:03|
|1  |2023-02-10 14:13:25.795|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  |2023-02-10 14:13:26.179|14  |13    |26     |
|1  |2023-02-10 14:13:26.179|14  |13    |26     |
+---+-----------------------+----+------+-------+
```


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.

countDistinct() – returns number of distinct elements in input column.

count() – returns number of elements in a column.

```python
from pyspark.sql.functions import approx_count_distinct, avg, round,
collect_list,collect_set,countDistinct

data1 = [('1', 'Sanjeevi', 'Male', 'IT','2500'), ('2', 'Josh', 'Female',
'CSE','3000'), ('10', 'San', 'Female', 'IT','2500')]
Schema1 = ['Id', 'Name', 'Gender', 'Dep','salary']

# Create DataFrame from data and schema
df = spark.createDataFrame(data1, Schema1)
df.show()

#Count Distinct
df.select(approx_count_distinct('Dep').alias('Dep_count')).show()

#Avg_function
df.select(round(avg('salary').alias('Avg_Salary'))).show()

#Collect_List --- it will give "array" type values with "duplicates"
df.select(collect_list('salary')).show()
```

```
+---+---------+------+---+------+
| Id|     Name|Gender|Dep|salary|
+---+---------+------+---+------+
|  1|Sanjeevi|  Male| IT|  2500|
|  2|     Josh|Female|CSE|  3000|
| 10|      San|Female| IT|  2500|
+---+---------+------+---+------+
```

```
+---------+
|Dep_count|
+---------+
|        2|
+---------+
```

```
+-----------------------------------+
|round(avg(salary) AS Avg_Salary, 0)|
+-----------------------------------+
|                             2667.0|
+-----------------------------------+
```