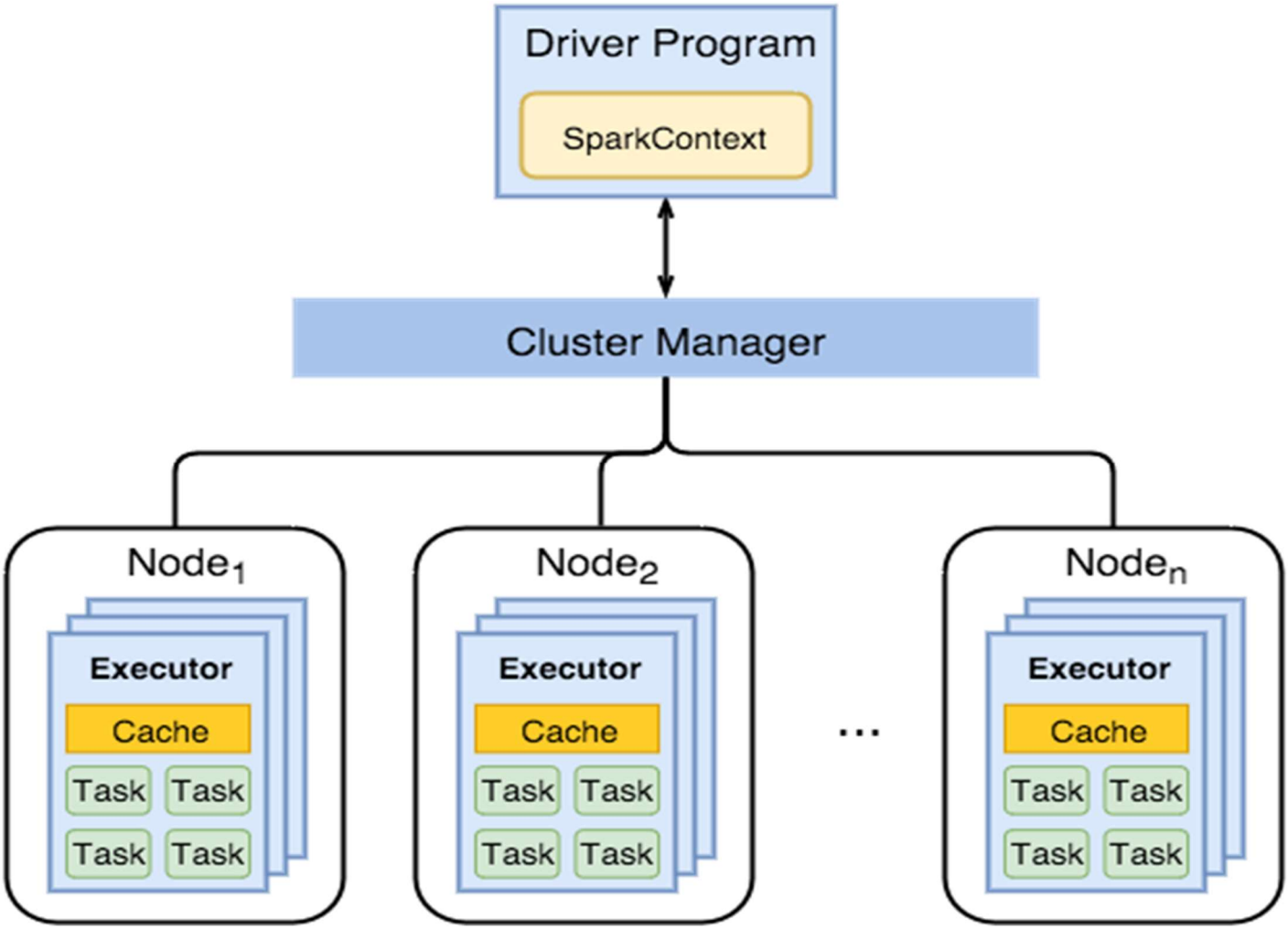
SPARK

Spark Components



1. Spark Driver

- *separate process to execute user applications*
- *creates SparkContext to schedule jobs execution and negotiate with cluster manager*

2. Executors

- *run tasks scheduled by driver*
- *store computation results in memory, on disk or off-heap*
- *interact with storage systems*

3. Cluster Manager

- *Mesos*
- *YARN*
- *Spark Standalone*

Spark Driver contains more components responsible for translation of user code into actual jobs executed on cluster:

- *SparkContext*

– *represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster*

- *DAGScheduler*

– *computes a DAG of stages for each job and submits them to TaskScheduler deter- mines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs*

- *TaskScheduler*

– *responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers*

- *SchedulerBackend*

– *backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local)*

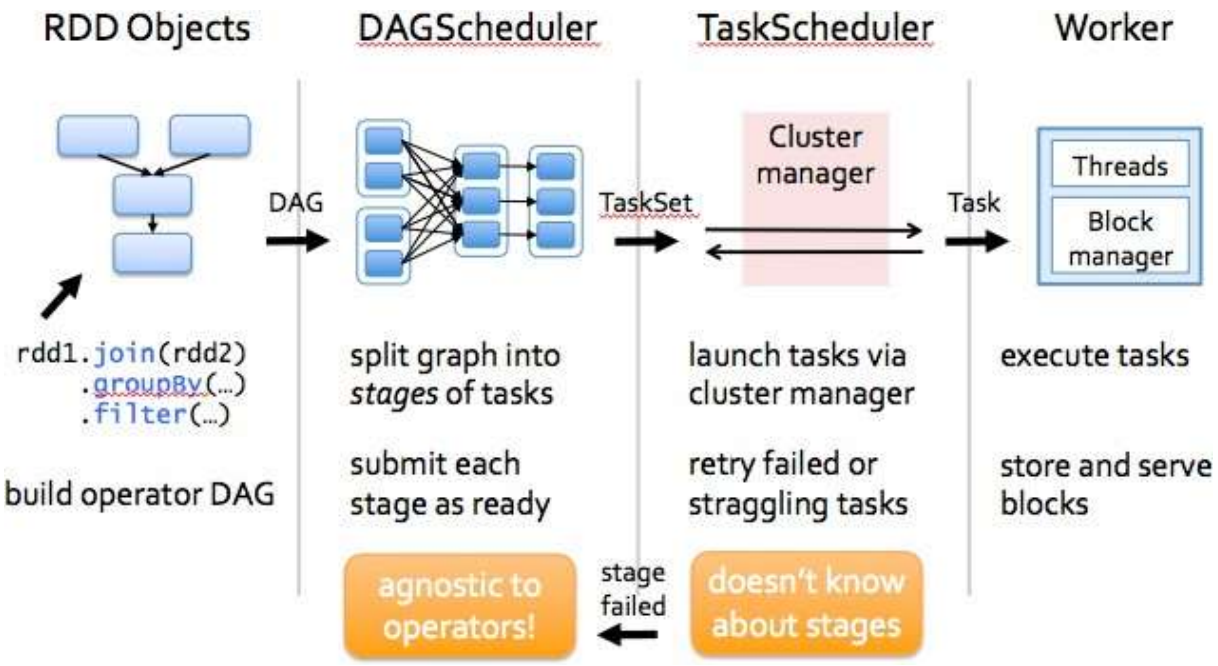
- *BlockManager*

– *provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)*

How Spark Works?

Spark has a small code base and the system is divided in various layers. Each layer has some responsibilities. The layers are independent of each other.

The first layer is the interpreter, Spark uses a Scala interpreter, with some modifications. As we enter wer code in spark console (creating RDD's and applying operators), Spark creates a operator graph. When the user runs an action (like collect), the Graph is submitted to a DAG Scheduler. The DAG scheduler divides operator graph into (map and reduce) stages. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together to optimize the graph. For e.g. Many map operators can be scheduled in a single stage. This optimization is key to Sparks performance. The final result of a DAG scheduler is a set of stages. The stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager. (Spark Standalone/Yarn/Mesos). The task scheduler doesn't know about dependencies among stages.



Create RDD

Usually, there are two popular ways to create the RDDs: loading an external dataset, or distributing a set of collection of objects. The following examples show some simplest ways to create RDDs by using parallelize() fucntion which takes an already existing collection in wer program and pass the same to the Spark Context.

1. *By using parallelize( ) function*

```
from pyspark.sql import SparkSession


spark = SparkSession \
    .builder \
    .appName("Python Spark create RDD example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

df = spark.sparkContext.parallelize([(1, 2, 3, 'a b c'),
                (4, 5, 6, 'd e f'),
                (7, 8, 9, 'g h i')]).toDF(['col1', 'col2', 'col3','col4'])
```

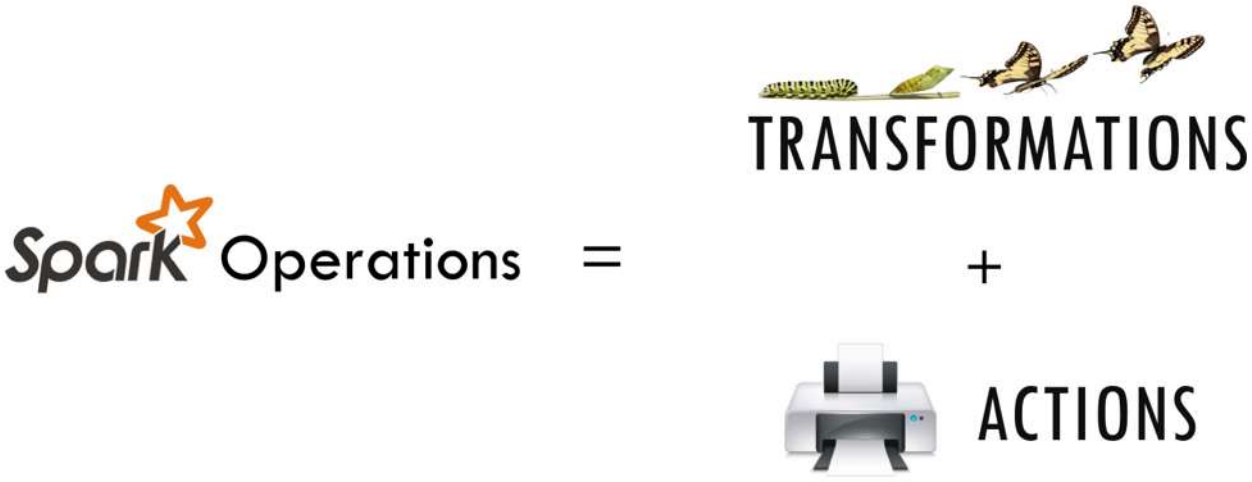
Then we will get the RDD data:

```
|col1|col2|col3| col4|
+----+----+----+-----+
|   1|   2|   3|a b c|
|   4|   5|   6|d e f|
|   7|   8|   9|g h i|
+----+----+----+-----+
```

Date : 31-07-2023

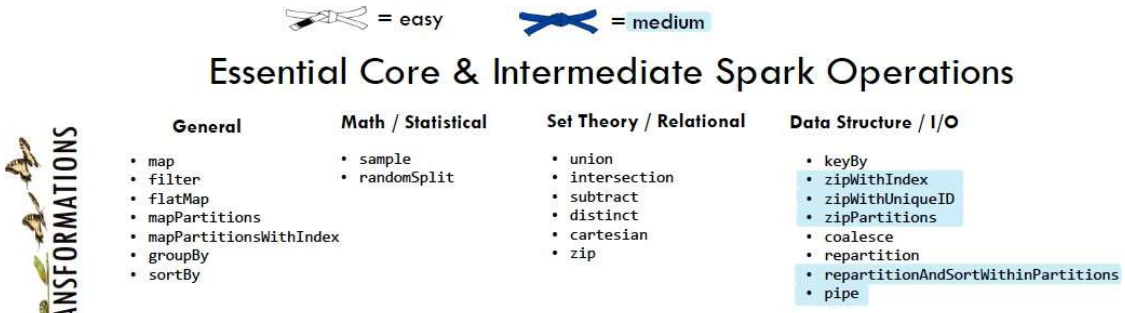
Assignments -7

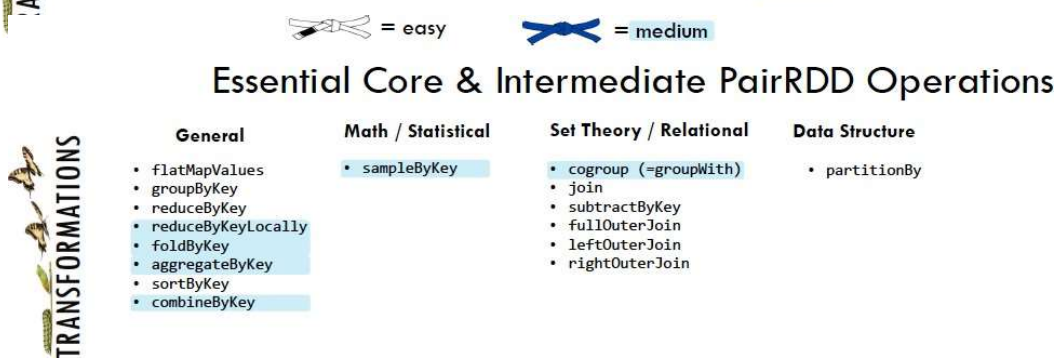
Spark Operations :



Transformations :

Transformations construct a new RDD from a previous one. For example, one common transformation isfiltering data that matches a predicate.





Actions :

Actions, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS).

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

- keys
- values

- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact

```
import pyspark
import findspark
from pyspark.sql import SparkSession
findspark.init()
spark = SparkSession.builder.appName("Nandha").getOrCreate()
data = [("Alice", 34), ("Bob", 45)]
df = spark.createDataFrame(data, ["Name", "Age"])
df.select("Name").show()
```

```
+-----+
| Name|
+-----+
|Alice|
|  Bob|
+-----+
```

```
from pyspark.sql import SparkSession

spark =SparkSession.builder.appName("group by key demo").getOrCreate()

sc=spark.sparkContext


#create an rdd

words =[("a",1),("b",1),("a",1),("b",1),("a",1),("c",3)]

myrdd =sc.parallelize(words)

myrdd.collect()

[('a', 1), ('b', 1), ('a', 1), ('b', 1), ('a', 1)]

mytrf= myrdd.groupByKey().mapValues(lambda x:list(x))

mytrf.collect()

[('a', [1, 1, 1]), ('b', [1, 1])]

myrByKeytrf =myrdd.reduceByKey(lambda x,y :x+y)

myrByKeytrf.collect()

[('a', 3), ('b', 2)]

myFoldByKey =myrdd.foldByKey(0,lambda x,y :x+y)

myFoldByKey.collect()

[('a', 12), ('b', 8)]
```

```
[('a', [1, 1, 1]), ('b', [1, 1])]
```

```
sorted(myrdd.countByKey().items())
```

```
[('a', 3), ('b', 2), ('c', 1)]
```

```
myrdd.cache()
```

```
ParallelCollectionRDD[51] at readRDDFromFile at PythonRDD.scala:274
```

```
myrdd.getStorageLevel()
```

```
StorageLevel(False, True, False, False, 1)
```

```
words1
=["january","february","march","april","may","may","april","march"]
words1rdd=sc.parallelize(words1)
```

```
words1rdd.collect()
```

```
['january', 'february', 'march', 'april', 'may', 'may', 'april',
'march']
```

```
student_scores = [("Alice", 85), ("Bob", 90), ("Alice", 95), ("Bob",
88), ("Charlie", 78)]

mapped_scores = sc.parallelize(student_scores).map(lambda x: (x[0],
x[1] + 5))
print(mapped_scores.collect())
```

```
filtered_scores = sc.parallelize(student_scores).filter(lambda x: x[1]
>= 90)
print(filtered_scores.collect())
```

```
flattened_names = sc.parallelize(student_scores).flatMap(lambda x:
list(x[0]))
print(flattened_names.collect())
```

```
grouped_scores = sc.parallelize(student_scores).groupBy(lambda x:
x[0])
print([(x, list(scores)) for x, scores in grouped_scores.collect()])
```

```
total_scores = sc.parallelize(student_scores).reduceByKey(lambda x, y:
x + y)
print(total_scores.collect())

sorted_scores = sc.parallelize(student_scores).sortBy(lambda x: x[1],
ascending=False)
print(sorted_scores.collect())


unique_names = sc.parallelize(student_scores).map(lambda x:
x[0]).distinct()
print(unique_names.collect())

additional_scores = [("David", 92), ("Eva", 87)]
combined_scores =
sc.parallelize(student_scores).union(sc.parallelize(additional_scores)
)
print(combined_scores.collect())


other_scores = [("Alice", 77), ("Charlie", 78)]
common_students =
sc.parallelize(student_scores).intersection(sc.parallelize(other_score
s))
print(common_students.collect())

student_info = [("Alice", "Physics"), ("Bob", "Math"), ("Charlie",
"History")]
combined_info =
sc.parallelize(student_scores).join(sc.parallelize(student_info))
print(combined_info.collect())


combined_info_left =
sc.parallelize(student_scores).leftOuterJoin(sc.parallelize(student_in
fo))
print(combined_info_left.collect())


additional_scores = [(75,), (95,)]
cartesian_product =
sc.parallelize(student_scores).cartesian(sc.parallelize(additional_sco
res))
print(cartesian_product.collect())
```

```
repartitioned_scores = sc.parallelize(student_scores).repartition(3)
print(repartitioned_scores.getNumPartitions())
```

```
[('Alice', 90), ('Bob', 95), ('Alice', 100), ('Bob', 93), ('Charlie',
83)]
[('Bob', 90), ('Alice', 95)]
['A', 'l', 'i', 'c', 'e', 'B', 'o', 'b', 'A', 'l', 'i', 'c', 'e', 'B',
'o', 'b', 'C', 'h', 'a', 'r', 'l', 'i', 'e']
[('Charlie', [('Charlie', 78)]), ('Bob', [('Bob', 90), ('Bob', 88)]),
('Alice', [('Alice', 85), ('Alice', 95)])]
[('Charlie', 78), ('Bob', 178), ('Alice', 180)]
[('Alice', 95), ('Bob', 90), ('Bob', 88), ('Alice', 85), ('Charlie',
78)]
['Charlie', 'Bob', 'Alice']
[('Alice', 85), ('Bob', 90), ('Alice', 95), ('Bob', 88), ('Charlie',
78), ('David', 92), ('Eva', 87)]
[('Charlie', 78)]
[('Charlie', (78, 'History')), ('Bob', (90, 'Math')), ('Bob', (88,
'Math')), ('Alice', (85, 'Physics')), ('Alice', (95, 'Physics'))]
[('Charlie', (78, 'History')), ('Bob', (90, 'Math')), ('Bob', (88,
'Math')), ('Alice', (85, 'Physics')), ('Alice', (95, 'Physics'))]
[(('Alice', 85), (75,)), (('Alice', 85), (95,)), (('Bob', 90), (75,)),
(('Bob', 90), (95,)), (('Alice', 95), (75,)), (('Alice', 95), (95,)),
(('Bob', 88), (75,)), (('Bob', 88), (95,)), (('Charlie', 78), (75,)),
(('Charlie', 78), (95,))]
3
```

```
other_scores = [("Alice", 77), ("Charlie", 78)]
common_students =
sc.parallelize(student_scores).intersection(sc.parallelize(other_score
s))
print(common_students.collect())
```

```
[('Charlie', 78)]
```

Date : 01-08-2023

Partitioning RDD and Acheiving Parallelism:

A Partition is a logical chunk of a large distributed data set.

Cache:

Caching is a way to persist an RDD in memory to reuse it across multiple Spark actions or transformations. When you cache an RDD, its partitions are stored in memory and can be quickly accessed without reevaluating the entire lineage.

Persistence:

Persistence is a more flexible version of caching, as it allows you to choose where to store the RDD: in memory, on disk, or a combination of both. PySpark provides several storage levels for persistence:

MEMORY_ONLY: Store RDD partitions in memory.

MEMORY_AND_DISK: Store RDD partitions in memory, and spill to disk if the memory is insufficient.

MEMORY_ONLY_SER: Store RDD partitions in memory as serialized Java objects.

MEMORY_AND_DISK_SER: Similar to MEMORY_AND_DISK, but stores serialized Java objects on disk if the memory is insufficient.

DISK_ONLY: Store RDD partitions on disk only.

Coalesce(numPartitions):

coalesce() is used to reduce the number of partitions in an RDD without shuffling the data. It merges existing partitions into a smaller number of partitions. This is an efficient operation when you want to reduce the number of partitions, especially after performing a filtering or repartitioning operation that leaves many small partitions.

Repartition(numPartitions):

repartition() is used to increase or decrease the number of partitions in an RDD. Unlike coalesce(), repartition() performs a full shuffle of the data across the specified number of partitions. This can be an expensive operation as it involves data movement across the cluster.

Glom():

glom() is a transformation that converts each partition of the RDD into a single list (array). It collapses each partition into a single element, where each element is an array representing the contents of the original partition. This can be useful when you want to operate on entire partitions at once or need to aggregate data at the partition level.

Narrow Transformations:

Narrow transformations are operations where each partition of the parent RDD contributes to only one partition of the resulting RDD. In other words, the data movement involved in narrow transformations is limited to the scope of individual partitions, and no data shuffling or exchange of data between partitions is required. Narrow transformations are performed in parallel within each partition and are more efficient because they don't involve costly data shuffling.

Examples of narrow transformations include map, filter, flatMap, union, mapPartitions, mapPartitionsWithIndex, filter, coalesce, etc.

Wide Transformations:

Wide transformations are operations where each partition of the parent RDD contributes to multiple partitions of the resulting RDD. In other words, wide transformations require data shuffling or redistribution across partitions, which involves significant data movement between nodes in the cluster. This can be more resource-intensive and time-consuming compared to narrow transformations.

Examples of wide transformations include groupBy, reduceByKey, sortByKey, join, cogroup, distinct, repartition, etc.

```
import pyspark as spark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("partitions in
pyspark").getOrCreate()

sc = spark.sparkContext

mylist=[1,2,3,4,5,6,7,8,9,11,12,13,14,15,16]

myrdd =sc.parallelize(mylist)

myrdd.getNumPartitions()

4

myrdd=myrdd.repartition(8)

myrdd.getNumPartitions()

8

myrdd=myrdd.coalesce(6)

myrdd.getNumPartitions()

6

myrdd.glom().collect()

[[7, 8, 9], [], [1, 2, 3, 4, 5, 6], [], [], [11, 12, 13, 14, 15, 16]]

sc = spark.sparkContext

rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.cache()

ParallelCollectionRDD[8] at readRDDFromFile at PythonRDD.scala:287

from pyspark import StorageLevel

rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.persist(StorageLevel.MEMORY_AND_DISK)

ParallelCollectionRDD[9] at readRDDFromFile at PythonRDD.scala:287

from pyspark import StorageLevel

rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.persist(StorageLevel.MEMORY_AND_DISK)

if rdd.is_cached:
    print("RDD is cached or persisted.")
else:
    print("RDD is not cached or persisted.")
```

Spark SQL :

Spark SQL is a component of Apache Spark that provides a programming interface for working with structured and semi-structured data using SQL queries, DataFrame APIs, and Dataset APIs. It allows users to perform SQL-like operations on distributed datasets, enabling seamless integration with traditional SQL-based data processing tools and supporting complex data processing tasks on large-scale data sets.

Features of Spark SQL:

Unified Data Processing: *Spark SQL provides a unified interface for querying structured and semi-structured data, including JSON, Parquet, ORC, Avro, and more. This makes it easier for developers and data engineers to work with different data formats within the same environment.*

Hive Compatibility: *Spark SQL is designed to be compatible with Apache Hive, which means you can use the same HiveQL queries and UDFs (User-Defined Functions) on Spark SQL. This facilitates seamless migration from Hive to Spark SQL and allows users to leverage their existing Hive knowledge and infrastructure.*

Performance Optimization: *Spark SQL leverages the Spark execution engine, which enables in-memory distributed data processing. This can lead to significant performance improvements over traditional Hive, especially for iterative and interactive data processing tasks.*

Integration with DataFrame/Dataset APIs: *Spark SQL integrates seamlessly with DataFrame and Dataset APIs, allowing users to mix SQL queries with functional programming constructs. This enables complex data transformations using a combination of SQL and functional programming.*

Catalog and Metastore Integration: *Spark SQL can integrate with external Hive metastores, Apache HCatalog, and other catalog providers. This makes it easier to share metadata across different Spark applications and traditional Hive workloads.*

Streaming Support: *Spark SQL extends its capabilities to support real-time data processing through Spark's Structured Streaming, enabling users to process data streams with SQL queries and combine batch and streaming data processing seamlessly.*

Advantages of Spark SQL over Hive:

Performance: *Spark SQL's in-memory processing and optimizations can lead to significantly faster query execution times compared to Hive, especially for iterative and interactive workloads.*

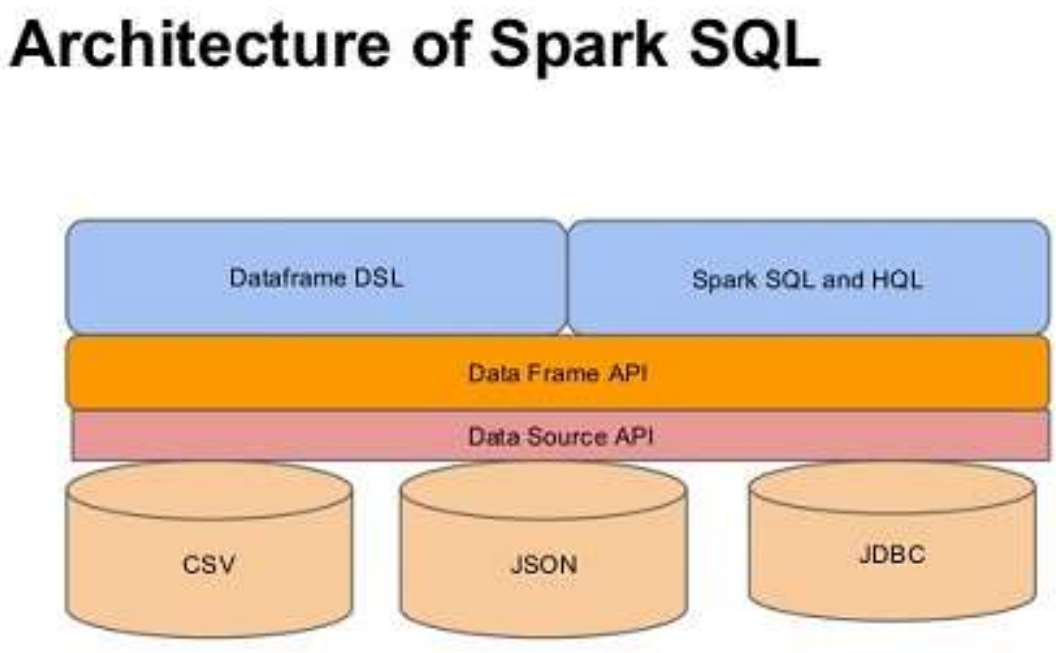
Unified API: *Spark SQL offers a unified programming API that includes SQL, DataFrame, and Dataset APIs, making it easier for developers to work with structured data and use SQL-like queries without the need to switch to different paradigms.*

Interactive Data Analysis: *Spark SQL provides an interactive shell (Spark SQL CLI) that allows users to explore data using SQL queries interactively. This is especially useful for ad-hoc data analysis and exploration.*

Flexible Data Sources: *Spark SQL supports a wide range of data formats, making it more versatile when dealing with different types of data sources compared to Hive.*

Integration with Spark Ecosystem: *Spark SQL is part of the larger Apache Spark ecosystem, enabling integration with other Spark components such as Spark MLlib, Spark GraphX, and more, facilitating a wide range of data processing tasks in a single framework.*

Ease of Use: *Spark SQL's compatibility with HiveQL allows users to leverage their existing Hive queries and knowledge while taking advantage of Spark's performance improvements.*



Data Source API: *The Data Source API in Spark SQL provides an interface for reading and writing data from various external sources like Parquet, Avro, JSON, etc., allowing seamless integration with different data formats and storage systems.*

Data Frame API and Interpreter/Optimizer: *The Data Frame API provides a higher-level abstraction over RDDs, enabling more convenient and efficient structured data manipulation. The Interpreter parses and compiles SQL or DataFrame operations, while the Optimizer, using Catalyst, applies rules to generate an optimized execution plan for better performance.Lazily Evaluated*

SQL Service: *The SQL Service in Spark SQL facilitates SQL query processing and execution on distributed datasets. It includes the parser to parse SQL queries, the Catalyst Optimizer to optimize query plans, and the Spark Execution Engine to execute queries across the Spark cluster.*

SQL CONTEXT :

SQLContext is a class and is used for initializing the functionalities of Spark SQL. SparkContext class object (sc) is required for initializing SQLContext class object. The following command is used for initializing the SparkContext through spark-shell. $ spark-shell.

Transformations and All actions:

I)Map

```
--student_scores = [("Alice", 85), ("Bob", 90), ("Alice", 95), ("Bob", 88), ("Charlie", 78)]

mapped_scores = sc.parallelize(student_scores).map(lambda x: (x[0], x[1] + 5))

print(mapped_scores.collect())
```

Output: [('Alice', 90), ('Bob', 95), ('Alice', 100), ('Bob', 93), ('Charlie', 83)]

II)Filter

```
--filtered_scores = sc.parallelize(student_scores).filter(lambda x: x[1] >= 90)

print(filtered_scores.collect())
```

Output: [('Bob', 90), ('Alice', 95)]

III)flatmap

flattened_names = sc.parallelize(student_scores).flatMap(lambda x: list(x[0]))

print(flattened_names.collect())

Output: ['A', 'l', 'i', 'c', 'e', 'B', 'o', 'b', 'A', 'l', 'i', 'c', 'e', 'B', 'o', 'b', 'C', 'h', 'a', 'r', 'l', 'i', 'e']

IV)Group-by

grouped_scores = sc.parallelize(student_scores).groupBy(lambda x: x[0])

print([(x, list(scores)) for x, scores in grouped_scores.collect()])

Output: [('Alice', [('Alice', 85), ('Alice', 95)]), ('Bob', [('Bob', 90), ('Bob', 88)]), ('Charlie', [('Charlie', 78)])]

v)ReduceByKey

total_scores = sc.parallelize(student_scores).reduceByKey(lambda x, y: x + y)

print(total_scores.collect())

Output: [('Alice', 180), ('Bob', 178), ('Charlie', 78)]

VI)SortBy

sorted_scores = sc.parallelize(student_scores).sortBy(lambda x: x[1], ascending=False)

print(sorted_scores.collect())

Output: [('Alice', 95), ('Bob', 90), ('Alice', 85), ('Bob', 88), ('Charlie', 78)]

VII)distinct

unique_names = sc.parallelize(student_scores).map(lambda x: x[0]).distinct()

print(unique_names.collect())

Output: ['Alice', 'Bob', 'Charlie']

VIII)Union

additional_scores = [("David", 92), ("Eva", 87)]

combined_scores = sc.parallelize(student_scores).union(sc.parallelize(additional_scores))

print(combined_scores.collect())

Output: [('Alice', 85), ('Bob', 90), ('Alice', 95), ('Bob', 88), ('Charlie', 78,('David',92),('Eva',87)]

IX)Intersection

other_scores = [("Alice", 77), ("Charlie", 78)]

common_students = sc.parallelize(student_scores).intersection(sc.parallelize(other_scores))

print(common_students.collect())

 # Output: [('Charlie', 78)]

X)Join

student_info = [("Alice", "Physics"), ("Bob", "Math"), ("Charlie", "History")]

combined_info = sc.parallelize(student_scores).join(sc.parallelize(student_info))

print(combined_info.collect())

Output: [('Alice', (85, 'Physics')), ('Bob', (90, 'Math')), ('Alice', (95, 'Physics')), ('Bob', (88, 'Math')), ('Charlie', (78, 'History'))]

XII)LeftOuterJoin

combined_info_left = sc.parallelize(student_scores).leftOuterJoin(sc.parallelize(student_info))

print(combined_info_left.collect())

 # Output: [('Alice', (85, 'Physics')), ('Bob', (90, 'Math')), ('Alice', (95, 'Physics')), ('Bob', (88, 'Math')), ('Charlie', (78, 'History'))]

 # ('Eva', (87, None)), ('David', (92, None))]

XIII)Cartesian

additional_scores = [(75,), (95,)]

cartesian_product = sc.parallelize(student_scores).cartesian(sc.parallelize(additional_scores))

print(cartesian_product.collect())

 # Output: [(('Alice', 85), (75,)), (('Alice', 85), (95,)), (('Bob', 90), (75,)), (('Bob', 90), (95,)), ...]

XIV)Repartition

repartitioned_scores = sc.parallelize(student_scores).repartition(3)

print(repartitioned_scores.getNumPartitions())

```python
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.appName("SimpleAverage").getOrCreate()

# Sample data: list of numbers
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Parallelize the data to create an RDD (Resilient Distributed
Dataset)
rdd = spark.sparkContext.parallelize(data)

# Calculate the average using PySpark functions
average = rdd.mean()

# Print the average
print("Average:",average)
```

```
Average: 5.5
```

```python
from pyspark.context import SparkContext
from pyspark.sql.session import SparkSession
sc= SparkContext.getOrCreate()
spark=SparkSession(sc)
```

```python
import pyspark
from pyspark.sql import SparkSession

spark =
SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
data = [("James","Smith","USA","CA"),
    ("Michael","Rose","USA","NY"),
    ("Robert","Williams","USA","CA"),
    ("Maria","Jones","USA","FL")
  ]
columns = ["firstname","lastname","country","state"]
df = spark.createDataFrame(data = data, schema = columns)
df.show(truncate=False)
```

```
+---------+--------+-------+-----+
|firstname|lastname|country|state|
+---------+--------+-------+-----+
|James    |Smith   |USA    |CA   |
|Michael  |Rose    |USA    |NY   |
|Robert   |Williams|USA    |CA   |
|Maria    |Jones   |USA    |FL   |
+---------+--------+-------+-----+
```

```
from pyspark.sql.functions import col
df.select(col("firstname"),col("lastname")).show()
```

```
+---------+--------+
|firstname|lastname|
+---------+--------+
|    James|   Smith|
|  Michael|    Rose|
|   Robert|Williams|
|    Maria|   Jones|
+---------+--------+
```

```
import pyspark
import findspark
from pyspark.sql import SparkSession
findspark.init()
spark = SparkSession.builder.appName("Nandha").getOrCreate()
data = [("Alice", 34), ("Bob", 45)]
df = spark.createDataFrame(data, ["Name", "Age"])
df.select("Name").show()
```

```
+-----+
| Name|
+-----+
|Alice|
|  Bob|
+-----+
```

```
df = spark.createDataFrame(
    [
        (1, "foo"),  # create your data here, be consistent in the
types.
        (2, "bar"),
    ],
    ["id", "label"]  # add your column names here
)

df.printSchema()

df.select("label").show()
```

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

+-----+
|label|
+-----+
|  foo|
```

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

```
# Create DataFrame in PySpark Shell
data = [("Java", "20000"), ("Python", "100000"), ("Scala", "3000")]
df = spark.createDataFrame(data)
df.show()
```

```
+------+------+
|    _1|    _2|
+------+------+
|  Java| 20000|
|Python|100000|
| Scala|  3000|
+------+------+
```

```
# Import PySpark
from pyspark.sql import SparkSession

#Create SparkSession
spark =
SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

# Data
data = [("Java", "20000"), ("Python", "100000"), ("Scala", "3000")]

# Columns
columns = ["language","users_count"]

# Create DataFrame
df = spark.createDataFrame(data).toDF(*columns)

# Print DataFrame
df.show()
```

```
+--------+-----------+
|language|users_count|
+--------+-----------+
|    Java|      20000|
|  Python|     100000|
|   Scala|       3000|
+--------+-----------+
```

```
from pyspark.sql import SparkSession
```

```
#creating spark session object
spark=SparkSession.builder.appName("Pyspark
Introduction").getOrCreate()

sc=spark.sparkContext

mylist=[1,2,3,4,5,6,7,8]

#create a rdd of a collection object
myrdd = sc.parallelize(mylist)

myrdd.collect()
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
mylist=[1,2,3,4,5,6,7,8]
#create a rdd of a collection object
myrdd = sc.parallelize(mylist)
#mytextrdd = sc.textFile("filepath")
myanotherrdd = myrdd.map(lambda x:x+1)
myanotherrdd.collect()
```

```
[2, 3, 4, 5, 6, 7, 8, 9]
```

```
data="Hello, Welcome to Edureka!".split(" ")
modified_data=sc.parallelize(data,2)
modified_data.collect()
sc.stop()
```

Spark SQL Libraries:

Spark SQL libraries are very useful as they interact with relational and procedural processing for managing the data frames. The libraries of Spark SQL are as follows:

1. DataFrame API:

DataFrame is a distributed collection of data where you will find the columns listed in an organized form. This is similar to the optimization techniques used in relational tables. An array of sources like Hive Tables, external databases, RDDs, data files, etc., can be used to construct a DataFrame.

2. SQL Interpreter and Optimizer:

The SQL interpreter and optimizer depend on the functional programming that can be done in the Scala programming language. As it is the most helpful component of SparkSQL, it provides the framework that can be used to transform trees, graphs, etc. This transformed data is useful for analysis, planning, optimization, and code-spawning at run time.

3. Data Source API:

Data Source API is the Universal API for fetching structured data from the data sources. The features of this API are as follows:
It supports various data sources such as Avro files, Hive databases, Parquet files, JSON documents, JDBC, etc.
It also supports smart sources of data.
You can also integrate it with third-party packages of Spark.
Data Source API can easily integrate with Big Data tools and frameworks through Spark-core.
It provides API for programming languages like Python, Scala, Java, and R.
It can process the data in massive amounts like the size of kilobytes or Petabytes.
The data processing can be done on a single node or multiple node clusters based on the data size.
This API works for data abstraction and domain-specific language for structured and semi-structured data.

4. SQL Service:

To work with structured data in Spark SQL, the SQL service is the first step you need to do. You can create DataFrame objects with the help of SQL service. You can also execute the SQL queries by using this library.

```python
#Rdd to DAtaframe
from pyspark.sql import SparkSession
spark=SparkSession.builder.appName("Rdd to dataframe").getOrCreate()

mylist=[(1, "Rocket", 30), (2, "Flight", 22), (3, "Jet", 23)]
myrdd = spark.sparkContext.parallelize(mylist)

from pyspark.sql.types import *

myschema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
])

df = spark.createDataFrame(myrdd, myschema)

df.show()
```

```
+---+------+---+
| id|  name|age|
+---+------+---+
|  1|Rocket| 30|
|  2|Flight| 22|
|  3|   Jet| 23|
+---+------+---+
```

```python
from pyspark.sql.functions import col,expr
data=[("2019-01-23",1),("2019-06-24",1)]

spark.createDataFrame(data).toDF("date","increment").select(col("date"
),col("increment"),expr("add_months(to_date(date,'yyyy-mm-
dd'),cast(increment as int))").alias("inc_date")).show()
```

```
+----------+---------+----------+
|      date|increment|  inc_date|
+----------+---------+----------+
|2019-01-23|        1|2019-02-23|
|2019-06-24|        1|2019-02-24|
+----------+---------+----------+
```

```python
from pyspark.sql.functions import *
data = [("1","2019-07-01"),("2","2019-06-24"),("3","2019-08-24")]
df=spark.createDataFrame(data=data,schema=["id","date"])

df.select(
      col("date"),
      current_date().alias("current_date"),
```

```
      datediff(current_date(),col("date")).alias("datediff")
    ).show()

+----------+------------+--------+
|      date|current_date|datediff|
+----------+------------+--------+
|2019-07-01|  2023-08-16|    1507|
|2019-06-24|  2023-08-16|    1514|
|2019-08-24|  2023-08-16|    1453|
+----------+------------+--------+


from pyspark.sql.functions import *
df.withColumn("datesDiff", datediff(current_date(),col("date"))) \
  .withColumn("montsDiff", months_between(current_date(),col("date")))
\
  .withColumn("montsDiff_round",round(months_between(current_date(),co
l("date")),2)) \
  .withColumn("yearsDiff",months_between(current_date(),col("date"))/
lit(12)) \
  .withColumn("yearsDiff_round",round(months_between(current_date(),co
l("date"))/lit(12),2)) \
  .show()

+---+----------+---------+-----------+--------------
+------------------+---------------+
| id|      date|datesDiff|  montsDiff|montsDiff_round|
yearsDiff|yearsDiff_round|
+---+----------+---------+-----------+--------------
+------------------+---------------+
|  1|2019-07-01|     1507|49.48387097|          49.48|
4.123655914166666|           4.12|
|  2|2019-06-24|     1514|49.74193548|          49.74|
4.14516129|           4.15|
|  3|2019-08-24|     1453|47.74193548|          47.74|
3.9784946233333334|           3.98|
+---+----------+---------+-----------+--------------
+------------------+---------------+


from pyspark import SparkContext, SparkConf
from pyspark.sql.types import StructType, StructField, StringType,
IntegerType, BooleanType, FloatType, ArrayType
from pyspark.sql.functions import to_date,avg
from pyspark.sql import Row

import pyspark
import findspark
from pyspark.sql import SparkSession
findspark.init()
spark = SparkSession.builder.appName("Santhu").getOrCreate()
```

```
sc=spark.sparkContext

json_data = [
    {"name": "John", "dob": "1990-01-01", "age": 31, "isGraduated":
True, "salary": 50000.0},
    {"name": "Alice", "dob": "1985-05-15", "age": 36, "isGraduated":
False, "salary": 60000.0},
    {"name": "Bob", "dob": "2000-11-20", "age": 21, "isGraduated":
True, "salary": 45000.0},
    {"name": "Eve", "dob": "1978-08-10", "age": 48, "isGraduated":
True, "salary": 75000.0},
]

# Define the schema for the DataFrame
schema = StructType([
    StructField("name", StringType(), True),
    StructField("dob", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("isGraduated", BooleanType(), True),
    StructField("salary", FloatType(), True),
])


# Create a DataFrame from the JSON data and specified schema
df = spark.createDataFrame(json_data, schema)

df = df.withColumn("dob", to_date(df["dob"], "yyyy-MM-dd"))

df.show()

+-----+----------+---+-----------+-------+
| name|       dob|age|isGraduated| salary|
+-----+----------+---+-----------+-------+
| John|1990-01-01| 31|       true|50000.0|
|Alice|1985-05-15| 36|      false|60000.0|
|  Bob|2000-11-20| 21|       true|45000.0|
|  Eve|1978-08-10| 48|       true|75000.0|
+-----+----------+---+-----------+-------+


data1 = [
    {"name": "John", "age": 30, "hobbies": ["Reading", "Cooking"]},
    {"name": "Alice", "age": 25, "hobbies": ["Singing", "Dancing",
"Painting"]},
    {"name": "Bob", "age": 28, "hobbies": ["Gardening"]},
    {"name": "Eve", "age": 33, "hobbies": []},
]

schema1 = StructType([
    StructField("name", StringType(), True),
```

```
    StructField("age", IntegerType(), True),
    StructField("hobbies", ArrayType(StringType()), True),
])

df1 = spark.createDataFrame(data1, schema1)

df1.show(truncate=False)
```

```
+-----+---+---------------------------+
|name |age|hobbies                    |
+-----+---+---------------------------+
|John |30 |[Reading, Cooking]         |
|Alice|25 |[Singing, Dancing, Painting]|
|Bob  |28 |[Gardening]                |
|Eve  |33 |[]                         |
+-----+---+---------------------------+
```

```
result_df = df1.filter("array_contains(hobbies, 'Dancing')")
result_df.show(truncate=False)
```

```
+-----+---+---------------------------+
|name |age|hobbies                    |
+-----+---+---------------------------+
|Alice|25 |[Singing, Dancing, Painting]|
+-----+---+---------------------------+
```

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('Broadcast.com').getOrCreate()

states = {"NY":"New York", "CA":"California", "FL":"Florida"}
broadcastStates = spark.sparkContext.broadcast(states)

data = [("James","Smith","USA","CA"),
    ("Michael","Rose","USA","NY"),
    ("Robert","Williams","USA","CA"),
    ("Maria","Jones","USA","FL")
  ]

rdd = spark.sparkContext.parallelize(data)

def state_convert(code):
    return broadcastStates.value[code]

result = rdd.map(lambda x:
(x[0],x[1],x[2],state_convert(x[3]))).collect()
print(result)
```

```
[('James', 'Smith', 'USA', 'California'), ('Michael', 'Rose', 'USA',
'New York'), ('Robert', 'Williams', 'USA', 'California'), ('Maria',
'Jones', 'USA', 'Florida')]
```

```
from pyspark.sql.functions import *
data4 = [
    ("Alice", "NY", 25),
    ("Bob", "CA", 32),
    ("John", "NY", 28),
    ("Mary", "CA", 40),
]

schema4 = ["name", "state", "age"]

df4 = spark.createDataFrame(data4, schema=schema4)

df4.show()
```

```
+-----+-----+---+
| name|state|age|
+-----+-----+---+
|Alice|   NY| 25|
|  Bob|   CA| 32|
| John|   NY| 28|
| Mary|   CA| 40|
+-----+-----+---+
```

```
grouped_df = df4.groupBy("state").agg(avg("age").alias("avg_age"),
count("*").alias("count"))


print("Grouped DataFrame:")
grouped_df.show()
```

```
Grouped DataFrame:
+-----+-------+-----+
|state|avg_age|count|
+-----+-------+-----+
|   NY|   26.5|    2|
|   CA|   36.0|    2|
+-----+-------+-----+
```

```
simpleData = [("James", "Sales", 3000),
    ("Michael", "Sales", 4600),
    ("Robert", "Sales", 4100),
    ("Maria", "Finance", 3000),
    ("James", "Sales", 3000),
```

```
    ("Scott", "Finance", 3300),
    ("Jen", "Finance", 3900),
    ("Jeff", "Marketing", 3000),
    ("Kumar", "Marketing", 2000),
    ("Saif", "Sales", 4100)
  ]
schema = ["employee_name", "department", "salary"]
df = spark.createDataFrame(data=simpleData, schema = schema)
df.printSchema()
df.show(truncate=False)
```

```
root
 |-- employee_name: string (nullable = true)
 |-- department: string (nullable = true)
 |-- salary: long (nullable = true)

+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|James        |Sales     |3000  |
|Michael      |Sales     |4600  |
|Robert       |Sales     |4100  |
|Maria        |Finance   |3000  |
|James        |Sales     |3000  |
|Scott        |Finance   |3300  |
|Jen          |Finance   |3900  |
|Jeff         |Marketing |3000  |
|Kumar        |Marketing |2000  |
|Saif         |Sales     |4100  |
+-------------+----------+------+
```

```
print("approx_count_distinct: " +
str(df.select(approx_count_distinct("salary")).collect()[0][0]))
```

```
approx_count_distinct: 6
```

```
print("avg: " + str(df.select(avg("salary")).collect()[0][0]))
```

```
avg: 3400.0
```

```
df.select(collect_list("salary")).show(truncate=False)
```

```
+------------------------------------------------------------+
|collect_list(salary)                                        |
+------------------------------------------------------------+
|[3000, 4600, 4100, 3000, 3000, 3300, 3900, 3000, 2000, 4100]|
+------------------------------------------------------------+
```

```
df.select(collect_set("salary")).show(truncate=False)
```

```
+-----------------------------------+
|collect_set(salary)                |
+-----------------------------------+
|[4600, 3000, 3900, 4100, 3300, 2000]|
+-----------------------------------+
```

```
df2 = df.select(countDistinct("department", "salary"))
df2.show(truncate=False)
print("Distinct Count of Department & Salary: "+str(df2.collect()[0]
[0]))
```

```
+---------------------------------+
|count(DISTINCT department, salary)|
+---------------------------------+
|8                                |
+---------------------------------+

Distinct Count of Department & Salary: 8
```

```
print("count: "+str(df.select(count("salary")).collect()[0]))
```

```
count: Row(count(salary)=10)
```

```
df.select(first("salary")).show(truncate=False)
```

```
+-------------+
|first(salary)|
+-------------+
|3000         |
+-------------+
```

```
df.select(last("salary")).show(truncate=False)
```

```
+------------+
|last(salary)|
+------------+
|4100        |
+------------+
```

```
df.select(kurtosis("salary")).show(truncate=False)
```

```
+-------------------+
|kurtosis(salary)   |
+-------------------+
|-0.6467803030303032|
+-------------------+
```

```
df.select(max("salary")).show(truncate=False)
```

```
+-----------+
|max(salary)|
+-----------+
|4600       |
+-----------+
```

```
df.select(min("salary")).show(truncate=False)
```

```
+-----------+
|min(salary)|
+-----------+
|2000       |
+-----------+
```

```
df.select(mean("salary")).show(truncate=False)
```

```
+-----------+
|avg(salary)|
+-----------+
|3400.0     |
+-----------+
```

```
df.select(skewness("salary")).show(truncate=False)
```

```
+--------------------+
|skewness(salary)    |
+--------------------+
|-0.12041791181069564|
+--------------------+
```

```
df.select(stddev("salary"),
stddev_samp("salary"),stddev_pop("salary")).show(truncate=False)
```

```
+------------------+------------------+------------------+
|stddev_samp(salary)|stddev_samp(salary)|stddev_pop(salary)|
+------------------+------------------+------------------+
|765.9416862050705  |765.9416862050705  |726.636084983398  |
+------------------+------------------+------------------+
```

```
df.select(sum("salary")).show(truncate=False)
```

```
+-----------+
|sum(salary)|
+-----------+
|34000      |
+-----------+
```

```
df.select(sumDistinct("salary")).show(truncate=False)
```

```
+--------------------+
|sum(DISTINCT salary)|
+--------------------+
|20900               |
+--------------------+
```

```
df.select(variance("salary"),var_samp("salary"),var_pop("salary")).sho
w(truncate=False)
```

```
+-----------------+-----------------+---------------+
|var_samp(salary) |var_samp(salary) |var_pop(salary)|
+-----------------+-----------------+---------------+
|586666.6666666666|586666.6666666666|528000.0       |
+-----------------+-----------------+---------------+
```

```
simpleData = [("James","Sales","NY",90000,34,10000),
    ("Michael","Sales","NY",86000,56,20000),
    ("Robert","Sales","CA",81000,30,23000),
    ("Maria","Finance","CA",90000,24,23000),
    ("Raman","Finance","CA",99000,40,24000),
    ("Scott","Finance","NY",83000,36,19000),
    ("Jen","Finance","NY",79000,53,15000),
    ("Jeff","Marketing","CA",80000,25,18000),
    ("Kumar","Marketing","NY",91000,50,21000)
  ]

schema = ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data=simpleData, schema = schema)
df.printSchema()
df.show(truncate=False)
```

```
root
 |-- employee_name: string (nullable = true)
 |-- department: string (nullable = true)
 |-- state: string (nullable = true)
 |-- salary: long (nullable = true)
 |-- age: long (nullable = true)
 |-- bonus: long (nullable = true)
```

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|Maria        |Finance   |CA   |90000 |24 |23000|
```

```
df.groupBy("department").avg("salary").show(truncate=False)
```

```
+----------+-----------------+
|department|avg(salary)      |
+----------+-----------------+
|Sales     |85666.66666666667|
|Finance   |87750.0          |
|Marketing |85500.0          |
+----------+-----------------+
```

```
df.groupBy("department").mean("salary").show(truncate=False)
```

```
+----------+-----------------+
|department|avg(salary)      |
+----------+-----------------+
|Sales     |85666.66666666667|
|Finance   |87750.0          |
|Marketing |85500.0          |
+----------+-----------------+
```

```
df.groupBy("department","state").sum("salary","bonus").show(truncate=F
alse)
```

```
+----------+-----+-----------+----------+
|department|state|sum(salary)|sum(bonus)|
+----------+-----+-----------+----------+
|Sales     |NY   |176000     |30000     |
|Sales     |CA   |81000      |23000     |
|Finance   |CA   |189000     |47000     |
|Finance   |NY   |162000     |34000     |
|Marketing |NY   |91000      |21000     |
|Marketing |CA   |80000      |18000     |
+----------+-----+-----------+----------+
```

```
from pyspark.sql.functions import sum,avg,max
df.groupBy("department").agg(sum("salary").alias("sum_salary"),avg("sa
lary").alias("avg_salary"),sum("bonus").alias("sum_bonus"),max("bonus"
).alias("max_bonus")).show(truncate=False)
```

```
+----------+----------+-----------------+---------+---------+
|department|sum_salary|avg_salary       |sum_bonus|max_bonus|
+----------+----------+-----------------+---------+---------+
|Sales     |257000    |85666.66666666667|53000    |23000    |
|Finance   |351000    |87750.0          |81000    |24000    |
|Marketing |171000    |85500.0          |39000    |21000    |
+----------+----------+-----------------+---------+---------+
```

```
|Raman        |Finance   |CA   |99000 |40 |24000|
|Scott        |Finance   |NY   |83000 |36 |19000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
+-------------+----------+-----+------+---+-----+
```

```
df.groupBy("department").sum("salary").show(truncate=False)
```

```
+----------+-----------+
|department|sum(salary)|
+----------+-----------+
|Sales     |257000     |
|Finance   |351000     |
|Marketing |171000     |
+----------+-----------+
```

```
df.groupBy("department").count().show(truncate=False)
```

```
+----------+-----+
|department|count|
+----------+-----+
|Sales     |3    |
|Finance   |4    |
|Marketing |2    |
+----------+-----+
```

```
df.groupBy("department").min("salary").show(truncate=False)
```

```
+----------+-----------+
|department|min(salary)|
+----------+-----------+
|Sales     |81000      |
|Finance   |79000      |
|Marketing |80000      |
+----------+-----------+
```

```
df.groupBy("department").max("salary").show(truncate=False)
```

```
+----------+-----------+
|department|max(salary)|
+----------+-----------+
|Sales     |90000      |
|Finance   |99000      |
|Marketing |91000      |
+----------+-----------+
```

```
from pyspark.sql.functions import sum,avg,max
df.groupBy("department").agg(sum("salary").alias("sum_salary"),avg("sa
lary").alias("avg_salary"),sum("bonus").alias("sum_bonus"),max("bonus"
).alias("max_bonus")).where(col("sum_bonus") >=
50000).show(truncate=False)
```

```
+----------+----------+-----------------+---------+---------+
|department|sum_salary|avg_salary       |sum_bonus|max_bonus|
+----------+----------+-----------------+---------+---------+
|Sales     |257000    |85666.66666666667|53000    |23000    |
|Finance   |351000    |87750.0          |81000    |24000    |
+----------+----------+-----------------+---------+---------+
```

```
from pyspark.sql.functions import sum, col, desc
df.groupBy("state").agg(sum("salary").alias("sum_salary")).filter(col(
"sum_salary") > 100000).sort(desc("sum_salary")).show()
```

```
+-----+----------+
|state|sum_salary|
+-----+----------+
|   NY|    429000|
|   CA|    350000|
+-----+----------+
```

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

spark = SparkSession.builder.appName('Joins in Pyspark').getOrCreate()


emp = [(1,"Smith",-1,"2018","10","M",3000), \
    (2,"Rose",1,"2010","20","M",4000), \
    (3,"Williams",1,"2010","10","M",1000), \
    (4,"Jones",2,"2005","10","F",2000), \
    (5,"Brown",2,"2010","40","",-1), \
      (6,"Brown",2,"2010","50","",-1) \
  ]
empColumns = ["emp_id","name","superior_emp_id","year_joined", \
       "emp_dept_id","gender","salary"]

empDF = spark.createDataFrame(data=emp, schema = empColumns)
empDF.printSchema()
empDF.show(truncate=False)
```

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

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

+------+--------+---------------+-----------+-----------+------+------
+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|
+------+--------+---------------+-----------+-----------+------+------
+
|1     |Smith   |-1             |2018       |10         |M     |3000
|
|2     |Rose    |1              |2010       |20         |M     |4000
|
|3     |Williams|1              |2010       |10         |M     |1000
|
|4     |Jones   |2              |2005       |10         |F     |2000
|
|5     |Brown   |2              |2010       |40         |      |-1
|
|6     |Brown   |2              |2010       |50         |      |-1
|
+------+--------+---------------+-----------+-----------+------+------
+
```

```
dept = [("Finance",10), \
    ("Marketing",20), \
    ("Sales",30), \
    ("IT",40) \
  ]
deptColumns = ["dept_name","dept_id"]
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.printSchema()
deptDF.show(truncate=False)
```

```
root
 |-- dept_name: string (nullable = true)
 |-- dept_id: long (nullable = true)

+---------+-------+
|dept_name|dept_id|
+---------+-------+
|Finance  |10     |
|Marketing|20     |
```

```
|6     |Brown   |2              |2010       |50         |      |-1
|null     |null   |
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+


empDF.join(deptDF,empDF.emp_dept_id ==
deptDF.dept_id,"full").show(truncate=False)

+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|1     |Smith   |-1             |2018       |10         |M     |3000
|Finance  |10     |
|3     |Williams|1              |2010       |10         |M     |1000
|Finance  |10     |
|4     |Jones   |2              |2005       |10         |F     |2000
|Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000
|Marketing|20     |
|null  |null    |null           |null       |null       |null  |null
|Sales    |30     |
|5     |Brown   |2              |2010       |40         |      |-1
|IT       |40     |
|6     |Brown   |2              |2010       |50         |      |-1
|null     |null   |
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+


empDF.join(deptDF,empDF.emp_dept_id ==
deptDF.dept_id,"fullouter").show(truncate=False)

+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|1     |Smith   |-1             |2018       |10         |M     |3000
|Finance  |10     |
|3     |Williams|1              |2010       |10         |M     |1000
|Finance  |10     |
|4     |Jones   |2              |2005       |10         |F     |2000
|Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000
|Marketing|20     |
|null  |null    |null           |null       |null       |null  |null
```

```
|Sales    |30     |
|IT       |40     |
+---------+-------+
```

```
empDF.join(deptDF,empDF.emp_dept_id ==
deptDF.dept_id,"inner").show(truncate=False)
```

```
+------+--------+--------------+----------+----------+------+------
+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|dept_name|dept_id|
+------+--------+--------------+----------+----------+------+------
+---------+-------+
|1     |Smith   |-1             |2018       |10         |M     |3000
|Finance  |10     |
|3     |Williams|1              |2010       |10         |M     |1000
|Finance  |10     |
|4     |Jones   |2              |2005       |10         |F     |2000
|Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000
|Marketing|20     |
|5     |Brown   |2              |2010       |40         |      |-1
|IT       |40     |
+------+--------+--------------+----------+----------+------+------
+---------+-------+
```

```
empDF.join(deptDF,empDF.emp_dept_id ==
deptDF.dept_id,"outer").show(truncate=False)
```

```
+------+--------+--------------+----------+----------+------+------
+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|dept_name|dept_id|
+------+--------+--------------+----------+----------+------+------
+---------+-------+
|1     |Smith   |-1             |2018       |10         |M     |3000
|Finance  |10     |
|3     |Williams|1              |2010       |10         |M     |1000
|Finance  |10     |
|4     |Jones   |2              |2005       |10         |F     |2000
|Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000
|Marketing|20     |
|null  |null    |null           |null       |null       |null  |null
|Sales    |30     |
|5     |Brown   |2              |2010       |40         |      |-1
|IT       |40     |
```

```
|Sales    |30     |
|5     |Brown   |2              |2010       |40         |      |-1
|IT       |40     |
|6     |Brown   |2              |2010       |50         |      |-1
|null     |null   |
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
```

```
empDF.join(deptDF,empDF.emp_dept_id ==
deptDF.dept_id,"left").show(truncate=False)
```

```
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|1     |Smith   |-1             |2018       |10         |M     |3000
|Finance  |10     |
|3     |Williams|1              |2010       |10         |M     |1000
|Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000
|Marketing|20     |
|4     |Jones   |2              |2005       |10         |F     |2000
|Finance  |10     |
|6     |Brown   |2              |2010       |50         |      |-1
|null     |null   |
|5     |Brown   |2              |2010       |40         |      |-1
|IT       |40     |
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
```

```
empDF.join(deptDF,empDF.emp_dept_id ==
deptDF.dept_id,"leftouter").show(truncate=False)
```

```
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|1     |Smith   |-1             |2018       |10         |M     |3000
|Finance  |10     |
|3     |Williams|1              |2010       |10         |M     |1000
|Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000
|Marketing|20     |
|4     |Jones   |2              |2005       |10         |F     |2000
|Finance  |10     |
```

```
|6     |Brown   |2              |2010       |50         |      |-1
|null     |null   |
|5     |Brown   |2              |2010       |40         |      |-1
|IT       |40     |
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
```

```
empDF.join(deptDF,empDF.emp_dept_id ==
deptDF.dept_id,"right").show(truncate=False)
```

```
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|4     |Jones   |2              |2005       |10         |F     |2000
|Finance  |10     |
|3     |Williams|1              |2010       |10         |M     |1000
|Finance  |10     |
|1     |Smith   |-1             |2018       |10         |M     |3000
|Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000
|Marketing|20     |
|null  |null    |null           |null       |null       |null  |null
|Sales    |30     |
|5     |Brown   |2              |2010       |40         |      |-1
|IT       |40     |
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
```

```
empDF.join(deptDF,empDF.emp_dept_id ==
deptDF.dept_id,"rightouter").show(truncate=False)
```

```
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|4     |Jones   |2              |2005       |10         |F     |2000
|Finance  |10     |
|3     |Williams|1              |2010       |10         |M     |1000
|Finance  |10     |
|1     |Smith   |-1             |2018       |10         |M     |3000
|Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000
|Marketing|20     |
|null  |null    |null           |null       |null       |null  |null
```

```
|Sales    |30     |
|5     |Brown   |2              |2010       |40         |      |-1
|IT       |40     |
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
```

```
empDF.join(deptDF,empDF.emp_dept_id ==
deptDF.dept_id,"leftsemi").show(truncate=False)
```

```
+------+--------+---------------+-----------+-----------+------+------
+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|
+------+--------+---------------+-----------+-----------+------+------
+
|1     |Smith   |-1             |2018       |10         |M     |3000
|
|3     |Williams|1              |2010       |10         |M     |1000
|
|4     |Jones   |2              |2005       |10         |F     |2000
|
|2     |Rose    |1              |2010       |20         |M     |4000
|
|5     |Brown   |2              |2010       |40         |      |-1
|
+------+--------+---------------+-----------+-----------+------+------
+
```

```
empDF.join(deptDF,empDF.emp_dept_id ==
deptDF.dept_id,"leftanti").show(truncate=False)
```

```
+------+-----+---------------+-----------+-----------+------+------+
|emp_id|name |superior_emp_id|year_joined|emp_dept_id|gender|salary|
+------+-----+---------------+-----------+-----------+------+------+
|6     |Brown|2              |2010       |50         |      |-1    |
+------+-----+---------------+-----------+-----------+------+------+
```

```
empDF.alias("emp1").join(empDF.alias("emp2"), \
    col("emp1.superior_emp_id") == col("emp2.emp_id"),"inner") \
    .select(col("emp1.emp_id"),col("emp1.name"), \
      col("emp2.emp_id").alias("superior_emp_id"), \
      col("emp2.name").alias("superior_emp_name")) \
   .show(truncate=False)
```

```
+------+--------+---------------+-----------------+
|emp_id|name    |superior_emp_id|superior_emp_name|
```

```
+------+--------+---------------+-----------------+
|2     |Rose    |1              |Smith            |
|3     |Williams|1              |Smith            |
|4     |Jones   |2              |Rose             |
|5     |Brown   |2              |Rose             |
|6     |Brown   |2              |Rose             |
+------+--------+---------------+-----------------+
```

```
empDF.createOrReplaceTempView("EMP")
deptDF.createOrReplaceTempView("DEPT")
```

```
joinDF = spark.sql("select * from EMP e, DEPT d where e.emp_dept_id ==
d.dept_id") \
  .show(truncate=False)
```

```
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|1     |Smith   |-1             |2018       |10         |M     |3000
|Finance  |10     |
|3     |Williams|1              |2010       |10         |M     |1000
|Finance  |10     |
|4     |Jones   |2              |2005       |10         |F     |2000
|Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000
|Marketing|20     |
|5     |Brown   |2              |2010       |40         |      |-1
|IT       |40     |
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
```

```
joinDF2 = spark.sql("select * from EMP e INNER JOIN DEPT d ON
e.emp_dept_id == d.dept_id") \
  .show(truncate=False)
```

```
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
|1     |Smith   |-1             |2018       |10         |M     |3000
```

```
|Finance  |10     |
|3     |Williams|1              |2010       |10         |M     |1000
|Finance  |10     |
|4     |Jones   |2              |2005       |10         |F     |2000
|Finance  |10     |
|2     |Rose    |1              |2010       |20         |M     |4000
|Marketing|20     |
|5     |Brown   |2              |2010       |40         |      |-1
|IT       |40     |
+------+--------+---------------+-----------+-----------+------+------
+---------+-------+
```

```
nulldata=[
    {"name":"jeeva","age":22,"dept":None},
    {"name":"sandy","age":44,"dept":"cse"},
    {"name":None,"age":12,"dept":"ece"}
]

dataf=spark.createDataFrame(nulldata)

dataf.show()
```

```
+---+----+-----+
|age|dept| name|
+---+----+-----+
| 22|null|jeeva|
| 44| cse|sandy|
| 12| ece| null|
+---+----+-----+
```

```
dd=dataf.na.drop()

dd.show()
```

```
+---+----+-----+
|age|dept| name|
+---+----+-----+
| 44| cse|sandy|
+---+----+-----+
```

```
data = [
    {"name": "Alice", "age": 25, "dept": "HR", "salary": 50000.0},
    {"name": "Bob", "age": None, "dept": "Finance", "salary": None},
    {"name": None, "age": 28, "dept": None, "salary": 60000.0},
    {"name": "John", "age": 32, "dept": "IT", "salary": 75000.0},
{"name": "Eva", "age": 23, "dept": "Sales", "salary": 55000.0},
    {"name": "Michael", "age": 40, "dept": "Marketing", "salary":
80000.0},
    {"name": "Sophia", "age": None, "dept": "Finance", "salary":
```

```
70000.0},
    {"name": None, "age": 35, "dept": "Operations", "salary":
65000.0},
    {"name": "William", "age": 29, "dept": "IT", "salary": None},
    {"name": "Emma", "age": 27, "dept": None, "salary": 60000.0},
    {"name": "Liam", "age": 31, "dept": "HR", "salary": 52000.0},
    {"name": "Olivia", "age": 24, "dept": "Marketing", "salary":
75000.0},
    {"name": "Noah", "age": 26, "dept": "Sales", "salary": 57000.0},
    {"name": None, "age": 33, "dept": "Finance", "salary": 68000.0},
    {"name": "Ava", "age": 37, "dept": "Operations", "salary": None},
    {"name": "James", "age": None, "dept": None, "salary": 70000.0},
    {"name": "Isabella", "age": 22, "dept": "IT", "salary": 71000.0},
    {"name": "Mia", "age": 29, "dept": "Sales", "salary": 59000.0}

]

spark = SparkSession.builder.appName("assignment").getOrCreate()

ndf = spark.createDataFrame(data)

ndf.show()
```

```
+----+----------+--------+-------+
| age|      dept|    name| salary|
+----+----------+--------+-------+
|  25|        HR|   Alice|50000.0|
|null|   Finance|     Bob|   null|
|  28|      null|    null|60000.0|
|  32|        IT|    John|75000.0|
|  23|     Sales|     Eva|55000.0|
|  40| Marketing| Michael|80000.0|
|null|   Finance|  Sophia|70000.0|
|  35|Operations|    null|65000.0|
|  29|        IT| William|   null|
|  27|      null|    Emma|60000.0|
|  31|        HR|    Liam|52000.0|
|  24| Marketing|  Olivia|75000.0|
|  26|     Sales|    Noah|57000.0|
|  33|   Finance|    null|68000.0|
|  37|Operations|     Ava|   null|
|null|      null|   James|70000.0|
|  22|        IT|Isabella|71000.0|
|  29|     Sales|     Mia|59000.0|
+----+----------+--------+-------+
```

```
mean_salary = ndf.select(mean("salary")).collect()[0][0]

mn=int(mean_salary)
```

```
mean=ndf.na.fill(mn,subset=["salary"])

mean.show()
```

```
+----+----------+--------+-------+
| age|      dept|    name| salary|
+----+----------+--------+-------+
|  25|        HR|   Alice|50000.0|
|null|   Finance|     Bob|64466.0|
|  28|      null|    null|60000.0|
|  32|        IT|    John|75000.0|
|  23|     Sales|     Eva|55000.0|
|  40| Marketing| Michael|80000.0|
|null|   Finance|  Sophia|70000.0|
|  35|Operations|    null|65000.0|
|  29|        IT| William|64466.0|
|  27|      null|    Emma|60000.0|
|  31|        HR|    Liam|52000.0|
|  24| Marketing|  Olivia|75000.0|
|  26|     Sales|    Noah|57000.0|
|  33|   Finance|    null|68000.0|
|  37|Operations|     Ava|64466.0|
|null|      null|   James|70000.0|
|  22|        IT|Isabella|71000.0|
|  29|     Sales|     Mia|59000.0|
+----+----------+--------+-------+
```

```
avg_salary = ndf.select(avg("salary")).collect()[0][0]

ag=int(avg_salary)

averg=ndf.na.fill(ag,subset=["salary"])

averg.show()
```

```
+----+----------+--------+-------+
| age|      dept|    name| salary|
+----+----------+--------+-------+
|  25|        HR|   Alice|50000.0|
|null|   Finance|     Bob|64466.0|
|  28|      null|    null|60000.0|
|  32|        IT|    John|75000.0|
|  23|     Sales|     Eva|55000.0|
|  40| Marketing| Michael|80000.0|
|null|   Finance|  Sophia|70000.0|
|  35|Operations|    null|65000.0|
|  29|        IT| William|64466.0|
|  27|      null|    Emma|60000.0|
|  31|        HR|    Liam|52000.0|
|  24| Marketing|  Olivia|75000.0|
|  26|     Sales|    Noah|57000.0|
```

```
|  33|   Finance|    null|68000.0|
|  37|Operations|     Ava|64466.0|
|null|      null|   James|70000.0|
|  22|        IT|Isabella|71000.0|
|  29|     Sales|     Mia|59000.0|
+----+----------+--------+-------+
```

```
ndf.show()
```

```
+----+----------+--------+-------+
| age|      dept|    name| salary|
+----+----------+--------+-------+
|  25|        HR|   Alice|50000.0|
|null|   Finance|     Bob|   null|
|  28|      null|    null|60000.0|
|  32|        IT|    John|75000.0|
|  23|     Sales|     Eva|55000.0|
|  40| Marketing| Michael|80000.0|
|null|   Finance|  Sophia|70000.0|
|  35|Operations|    null|65000.0|
|  29|        IT| William|   null|
|  27|      null|    Emma|60000.0|
|  31|        HR|    Liam|52000.0|
|  24| Marketing|  Olivia|75000.0|
|  26|     Sales|    Noah|57000.0|
|  33|   Finance|    null|68000.0|
|  37|Operations|     Ava|   null|
|null|      null|   James|70000.0|
|  22|        IT|Isabella|71000.0|
|  29|     Sales|     Mia|59000.0|
+----+----------+--------+-------+
```

```
max_salary = ndf.select(max("salary")).collect()[0][0]

mx=int(max_salary)

max_s=ndf.na.fill(mx,subset=["salary"])

max_s.show()
```

```
+----+----------+--------+-------+
| age|      dept|    name| salary|
+----+----------+--------+-------+
|  25|        HR|   Alice|50000.0|
|null|   Finance|     Bob|80000.0|
|  28|      null|    null|60000.0|
|  32|        IT|    John|75000.0|
|  23|     Sales|     Eva|55000.0|
|  40| Marketing| Michael|80000.0|
```

```
|null|   Finance|  Sophia|70000.0|
|  35|Operations|    null|65000.0|
|  29|        IT| William|80000.0|
|  27|      null|    Emma|60000.0|
|  31|        HR|    Liam|52000.0|
|  24| Marketing|  Olivia|75000.0|
|  26|     Sales|    Noah|57000.0|
|  33|   Finance|    null|68000.0|
|  37|Operations|     Ava|80000.0|
|null|      null|   James|70000.0|
|  22|        IT|Isabella|71000.0|
|  29|     Sales|     Mia|59000.0|
+----+----------+--------+-------+
```

```
name = "NoName"
```

```
name_fill=ndf.na.fill(name,subset=["name"])
```

```
name_fill.show()
```

```
+----+----------+--------+-------+
| age|      dept|    name| salary|
+----+----------+--------+-------+
|  25|        HR|   Alice|50000.0|
|null|   Finance|     Bob|   null|
|  28|      null|  NoName|60000.0|
|  32|        IT|    John|75000.0|
|  23|     Sales|     Eva|55000.0|
|  40| Marketing| Michael|80000.0|
|null|   Finance|  Sophia|70000.0|
|  35|Operations|  NoName|65000.0|
|  29|        IT| William|   null|
|  27|      null|    Emma|60000.0|
|  31|        HR|    Liam|52000.0|
|  24| Marketing|  Olivia|75000.0|
|  26|     Sales|    Noah|57000.0|
|  33|   Finance|  NoName|68000.0|
|  37|Operations|     Ava|   null|
|null|      null|   James|70000.0|
|  22|        IT|Isabella|71000.0|
|  29|     Sales|     Mia|59000.0|
+----+----------+--------+-------+
```

```
dept = "unKnown"
```

```
dept_fill=ndf.na.fill(dept,subset=["dept"])
```

```
dept_fill.show()
```

```
+----+----------+--------+-------+
| age|      dept|    name| salary|
+----+----------+--------+-------+
|  25|        HR|   Alice|50000.0|
|null|   Finance|     Bob|   null|
|  28|   unKnown|    null|60000.0|
|  32|        IT|    John|75000.0|
|  23|     Sales|     Eva|55000.0|
|  40| Marketing| Michael|80000.0|
|null|   Finance|  Sophia|70000.0|
|  35|Operations|    null|65000.0|
|  29|        IT| William|   null|
|  27|   unKnown|    Emma|60000.0|
|  31|        HR|    Liam|52000.0|
|  24| Marketing|  Olivia|75000.0|
|  26|     Sales|    Noah|57000.0|
|  33|   Finance|    null|68000.0|
|  37|Operations|     Ava|   null|
|null|   unKnown|   James|70000.0|
|  22|        IT|Isabella|71000.0|
|  29|     Sales|     Mia|59000.0|
+----+----------+--------+-------+
```

```
from pyspark.sql import SparkSession
spark =
SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
data = [("James","M",60000),("Michael","M",70000),
        ("Robert",None,400000),("Maria","F",500000),
        ("Jen","",None)]

columns = ["name","gender","salary"]
df = spark.createDataFrame(data = data, schema = columns)
df.show()
```

```
+-------+------+------+
|   name|gender|salary|
+-------+------+------+
|  James|     M| 60000|
|Michael|     M| 70000|
| Robert|  null|400000|
|  Maria|     F|500000|
|    Jen|      |  null|
+-------+------+------+
```

```
from pyspark.sql.functions import when
df2 = df.withColumn("new_gender", when(df.gender == "M","Male")
                                  .when(df.gender == "F","Female")
                                  .when(df.gender.isNull() ,"")
```

```
                                    .otherwise(df.gender))
df2.show()
```

```
+-------+------+------+----------+
|   name|gender|salary|new_gender|
+-------+------+------+----------+
|  James|     M| 60000|      Male|
|Michael|     M| 70000|      Male|
| Robert|  null|400000|          |
|  Maria|     F|500000|    Female|
|    Jen|      |  null|          |
+-------+------+------+----------+
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('Map Partitions').getOrCreate()
data = [('James','Smith','M',3000),
  ('Anna','Rose','F',4100),
  ('Robert','Williams','M',6200),
]

columns = ["firstname","lastname","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
df.show()


def reformat(partitionData):
    for row in partitionData:
        yield [row.firstname+","+row.lastname,row.salary*10/100]
df2=df.rdd.mapPartitions(reformat).toDF(["name","bonus"])
df2.show()

#Example 2 mapPartitions()
def reformat2(partitionData):
  updatedData = []
  for row in partitionData:
    name=row.firstname+","+row.lastname
    bonus=row.salary*10/100
    updatedData.append([name,bonus])
  return iter(updatedData)

df2=df.rdd.mapPartitions(reformat).toDF(["name","bonus"])
df2.show()
```

```
+---------+--------+------+------+
|firstname|lastname|gender|salary|
+---------+--------+------+------+
|    James|   Smith|     M|  3000|
|     Anna|    Rose|     F|  4100|
```

```
|   Robert|Williams|     M|  6200|
+---------+--------+------+------+

+---------------+-----+
|           name|bonus|
+---------------+-----+
|    James,Smith|300.0|
|      Anna,Rose|410.0|
|Robert,Williams|620.0|
+---------------+-----+

+---------------+-----+
|           name|bonus|
+---------------+-----+
|    James,Smith|300.0|
|      Anna,Rose|410.0|
|Robert,Williams|620.0|
+---------------+-----+
```

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('pyspark-by-
examples').getOrCreate()

arrayArrayData = [
  ("James",[["Java","Scala","C++"],["Spark","Java"]]),
  ("Michael",[["Spark","Java","C++"],["Spark","Java"]]),
  ("Robert",[["CSharp","VB"],["Spark","Python"]])
]

df = spark.createDataFrame(data=arrayArrayData, schema =
['name','subjects'])
df.printSchema()
df.show(truncate=False)
```

```
root
 |-- name: string (nullable = true)
 |-- subjects: array (nullable = true)
 |    |-- element: array (containsNull = true)
 |    |    |-- element: string (containsNull = true)

+-------+-----------------------------------+
|name   |subjects                           |
+-------+-----------------------------------+
|James  |[[Java, Scala, C++], [Spark, Java]]|
|Michael|[[Spark, Java, C++], [Spark, Java]]|
|Robert |[[CSharp, VB], [Spark, Python]]    |
```

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

```
from pyspark.sql.functions import explode
df.select(df.name,explode(df.subjects)).show(truncate=False)
```

```
+-------+------------------+
|name   |col               |
+-------+------------------+
|James  |[Java, Scala, C++]|
|James  |[Spark, Java]     |
|Michael|[Spark, Java, C++]|
|Michael|[Spark, Java]     |
|Robert |[CSharp, VB]      |
|Robert |[Spark, Python]   |
+-------+------------------+
```

```
from pyspark.sql.functions import flatten
df.select(df.name,flatten(df.subjects)).show(truncate=False)
```

```
+-------+------------------------------+
|name   |flatten(subjects)             |
+-------+------------------------------+
|James  |[Java, Scala, C++, Spark, Java]|
|Michael|[Spark, Java, C++, Spark, Java]|
|Robert |[CSharp, VB, Spark, Python]    |
+-------+------------------------------+
```

```
from pyspark.sql.types import *

data = [
    [101, 'John Doe', ['Math', 'Science', 'English'], ['History',
'Geography']],
    [102, 'Jane Smith', ['Physics', 'Chemistry', 'Biology'],
['Literature', 'Art']],
    [103, 'Michael Lee', ['Computer Science', 'Math', 'Physics'],
['History', 'English']],
    [104, 'Emily Chen', ['English', 'Biology', 'Art'], ['Physics',
'Chemistry']],
    [105, 'Robert Kim', ['History', 'Geography', 'Economics'],
['Math', 'Science']],
]

schema = StructType([
    StructField("ID", IntegerType(), False),
    StructField("Name", StringType(), False),
    StructField("GoodSubjects", ArrayType(StringType(), False),
False),
    StructField("BadSubjects", ArrayType(StringType(), False), False)
```

```
])

# Create the DataFrame
df = spark.createDataFrame(data, schema)
df.show(truncate=False)

+---+-----------+---------------------------------
+--------------------+
|ID |Name       |GoodSubjects                     |BadSubjects
|
+---+-----------+---------------------------------
+--------------------+
|101|John Doe   |[Math, Science, English]         |[History,
Geography]|
|102|Jane Smith |[Physics, Chemistry, Biology]    |[Literature, Art]
|
|103|Michael Lee|[Computer Science, Math, Physics]|[History, English]
|
|104|Emily Chen |[English, Biology, Art]          |[Physics,
Chemistry]|
|105|Robert Kim |[History, Geography, Economics]  |[Math, Science]
|
+---+-----------+---------------------------------
+--------------------+


combined_df = df.select("ID", "Name",
flatten(expr("array(GoodSubjects,
BadSubjects)")).alias("AllSubjects"))

---------------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
<ipython-input-17-9d394555499d> in <module>
----> 1 combined_df = df.select("ID", "Name",
flatten(expr("array(GoodSubjects,
BadSubjects)")).alias("AllSubjects"))

NameError: name 'expr' is not defined
```

Spark Streaming Data:

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.

Spark Streaming provides a high-level abstraction called discretized stream *or* DStream*, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.*

Dealing with multiple nested levels :

To make multiple arrays in single array: ***Use flatten function.***

If i want to break an internal structure: ***Use the explode function.***

#For Real-Time Data Monitoring:

- *Real time data works on single record.*
- *Real-time , Near real time processing is really fast*

Solution is ***Spark streaming***

*Series of Continuous RDDs will be called **DStreams**[multiple unbounded RDDs]*

Fundamental Unit is Dstream:

You can apply Pyspark sql logic over Dstreams to play with the transformations.-->Create new RDD after transformation.

It has a capability of reading data from static data[RDBMS] and streaming data also.

Benefit of Spark Streaming :

- *Achieves Low Latency*
- *Track behavior of customers*
- *Scales to hundreds of nodes*
- *Integrates with batch and real-time processing*

ex: Uber Streaming

Discretized Stream(Dstream):

- *It received from source or from a processed data stream generated by transforming the input stream.*
- *Internally, Dstream is represented by a continuous stream of RDDs.*
- *Basic Sources like File Systems, Socket Connections.*
- *Advanced Source Kafka, Flume, Kinesis.*
- *Read your data with Network Port also.*
- *Can Lock a port no. and generated the data manually continously,spark has to read data and graph that data:Socket Conenction(You can grab from FLUME).*

#Structured Streaming:

Transformations on DStream:

To break the DAG[Output operation]

-print()

-saveAsTextFiles()

-saveAsObjectFiles()--Java

-saveAsHadoopFiles--Seq file

-foreach

--Cache/Persist

-------Broadcast Variables allows programemr to keepa raed only variabel cachjed on each amchine rather than shiiping a copy of it with tasks

---They can eb sued to give every node a copy of a large input dataset in an efficient manner.

Spark also attempts to distribute broadcast varuables using efficient broadcast algos to reduce communciation cost.

#Checkpoints

#Netcat si a data source generator-we lock a port so that we can send the data to spark stream

#To write a spark streaming Job:

#spark streaming Job

from pyspark.sql import SparkSession

from pyspark.streaming import StreamingContext

#spark session object

spark = sparkSession.builder.appName("absvdc").getOrCreate()

sc=spark.sparkContext

#streaming Context

ssc = StreamingContext(sc,10)

#DStreams

mydstream = ssc.socketTextStream("localhost" , 12345)

#Transformations

mywords = mydstream.flatMap(lambda x: x.split(" ")).map(lambda x: (x, 1)

mywordcount = mywords.reduceByKey(lambda x,y: x + y)

#Call output operation

mywordcount.pprint()

#To receive input data,start your stream

ssc.start()

ssc.awaitTermination()

#Stop streaming

ssc.stop()

#In strcutured streaming we are converting,Instead of Dstream we are creating Streaming DataFrame-

#Create sessionobejct

from pyspark.sql import SparkSession

*from pyspark.streaming import **

spark = SparkSession.builder.appName().getOrCreate()

mystreamingdf = spark.readStream.format("socket").option("host", "localhost").option("port" , 12345).load()

Output format :

overwrite

append

complete

 # Structured Streaming

from pyspark.sql import SparkSession

*from pyspark.streaming import **

*from pyspark.sql.functions import **

spark = SparkSession.builder.appName("chet app").getOrCreate()

mystreamingdf = spark.readStream.format("socket").option("host", "localhost").option("port" , 12348).load()

mywordcount = mystreamingdf.select(explode(split("value" , " ")).alias("word"))

countdf = mywordcount.groupBy("word").count()

word_write = countdf.writeStream.format("console").outputMode("complete").start().awaitTermination()

#Reading From FileSystem-

from pyspark.sql import SparkSession

*from pyspark.streaming import **

*from pyspark.sql.functions import **

*from pyspark.sql.types import **

spark = SparkSession.builder.appName("Structured Streaming").master("local[*]").getOrCreate()

spark.sparkContext.setLogLevel("ERROR")

myschema = StructType().add("name","string").add("Occupation","string").add("age","integer")

filesource = spark.readStream.csv("samplecsv.csv" , schema = myschema)

agecount = filesource.groupBy("name"),count()

agecount.writeStream.format("console").outputMode("complete").start().awaitTermination()