Apache SPARK

Notes by Mannan Ul Haq (BDS-4A)

Definition: Open-source cluster computing framework for handling real-time generated data.

Optimization: Built on top of Hadoop MapReduce, Spark processes data in memory for faster processing compared to disk-based alternatives.

History:

- Initiated by Matei Zaharia at UC Berkeley's AMPLab in 2009.
- Open-sourced in 2010 under BSD license.
- Acquired by Apache Software Foundation in 2013.
- Emerged as a Top-Level Apache Project in 2014.

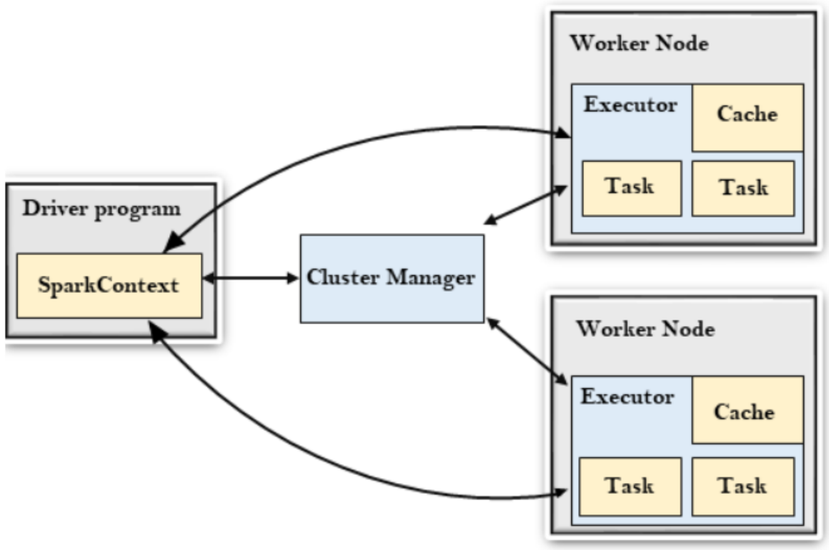
Features:

- High performance for both batch and streaming data.
- Supports Java, Scala, Python, R, and SQL with over 80 high-level operators.
- Libraries include SQL, DataFrames, MLlib, GraphX, and Spark Streaming.
- Unified analytics engine for large-scale data processing.
- Runs on Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud.

Spark Architecture

Master-Slave Architecture: Cluster comprises a single master and multiple slaves.

1. **Resilient Distributed Dataset (RDD)**
    - Group of data items stored in-memory on worker nodes.
    - **Resilient**: Data recovery on failure.
    - **Distributed**: Data distributed among nodes.
    - **Dataset**: Collection of data.
2. **Directed Acyclic Graph (DAG)**
    - Finite direct graph performing sequence of computations on data.
    - Each node represents an RDD partition; edges represent transformations on data.

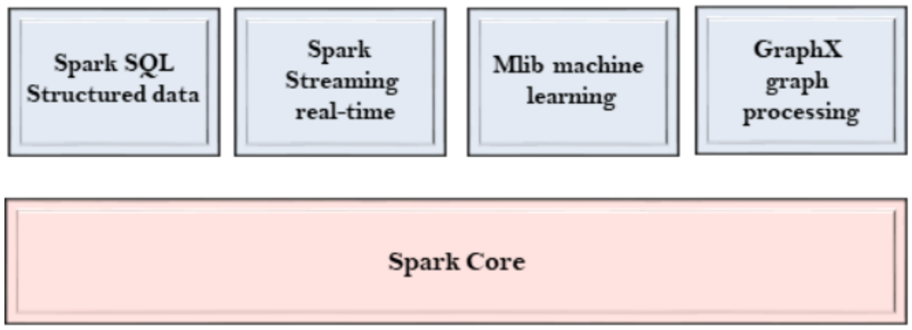


Components of Spark Architecture

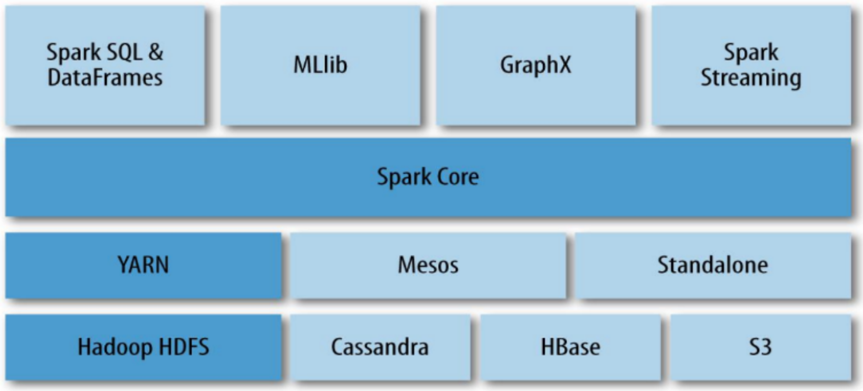
1. **Driver Program**
    - Process executing main() function, creating SparkContext object.
    - SparkContext coordinates Spark applications on cluster.
    - Tasks:
        - Acquiring executors on cluster nodes.
        - Sending application code to executors (via JAR or Python files).
        - Sending tasks to executors for execution.
2. **Cluster Manager**
    - Allocates resources across applications.
    - Types: Hadoop YARN, Apache Mesos, Standalone Scheduler.
    - Standalone Scheduler: Installs Spark on empty set of machines.
3. **Worker Node**
    - Slave node running application code in cluster.
4. **Executor**
    - Process launched for application on worker node.
    - Runs tasks, stores data in memory/disk across them.
    - Reads/writes data to external sources.
    - Each application contains its executor.
5. **Task**

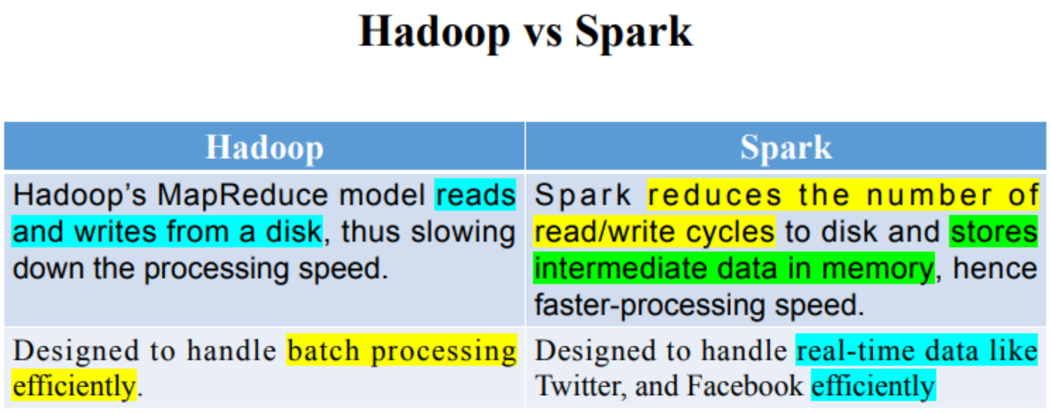
- Unit of work sent to one executor.

Spark Components



1. **Spark Core**
    - Core functionality of Spark.
    - Components:
        - Task scheduling.
        - Fault recovery.
        - Interaction with storage systems.
        - Memory management.
2. **Spark SQL**
    - Built on Spark Core, supports structured data.
    - Features:
        - SQL and HQL (Hive Query Language) queries.
        - Supports various data sources like HDFS, Hive tables, JSON, Cassandra.20, HBase.
3. **Spark Streaming**
    - Supports scalable, fault-tolerant processing of streaming data.
    - Utilizes Spark Core for fast scheduling and real-time processing.
4. **MLlib**
    - Machine Learning library with various algorithms.
    - Includes correlations, classification, regression, clustering, PCA.
    - Nine times faster than disk-based implementations like Apache Mahout.
5. **GraphX**
    - Library for manipulating graphs and performing graph-parallel computations.





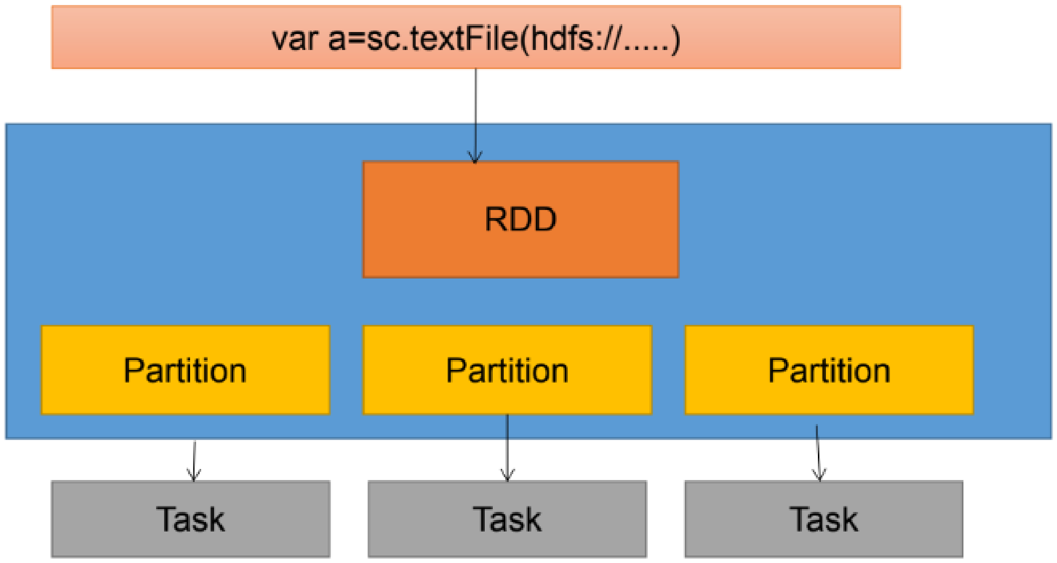
Hadoop vs Spark

Hadoop	Spark
Hadoop's MapReduce model reads and writes from a disk, thus slowing down the processing speed.	Spark reduces the number of read/write cycles to disk and stores intermediate data in memory, hence faster-processing speed.
Designed to handle batch processing efficiently.	Designed to handle real-time data like Twitter, and Facebook efficiently

What is RDD?

RDD stands for Resilient Distributed Dataset. It is a fundamental abstraction in Apache Spark that represents a collection of immutable, partitioned data distributed across a cluster of machines. Here's a breakdown of its key characteristics:

1. **Resilient**: RDDs are resilient because they can automatically recover from failures. Spark keeps track of the lineage of transformations applied to an RDD, so in case of a failure, it can reconstruct lost partitions by reapplying those transformations.
2. **Distributed**: RDDs distribute data across multiple nodes in a cluster, allowing parallel processing of data. Each RDD is divided into partitions, with each partition being processed by a separate task on different nodes in the cluster.
3. **Dataset**: RDD represents a collection of data elements that can be operated on in parallel.



Creating RDD:

There are two common ways to create RDDs: using the `parallelize()` method and using the `textFile()` method.

1. parallelize() Method

The `parallelize()` method is used to create an RDD from a collection of objects that exist on the driver node. This method takes a collection (such as a list or array) and distributes the data across the cluster to form an RDD.

Here's an example of using the `parallelize()` method in Python with PySpark:

```
# Create a collection of data
data = [1, 2, 3, 4, 5]

# Parallelize the collection to create an RDD
rdd = sc.parallelize(data)

# Collect the results back to the driver program
result = rdd.collect()
print(result)  # Output: [1, 2, 3, 4, 5]
```

2. textFile() Method

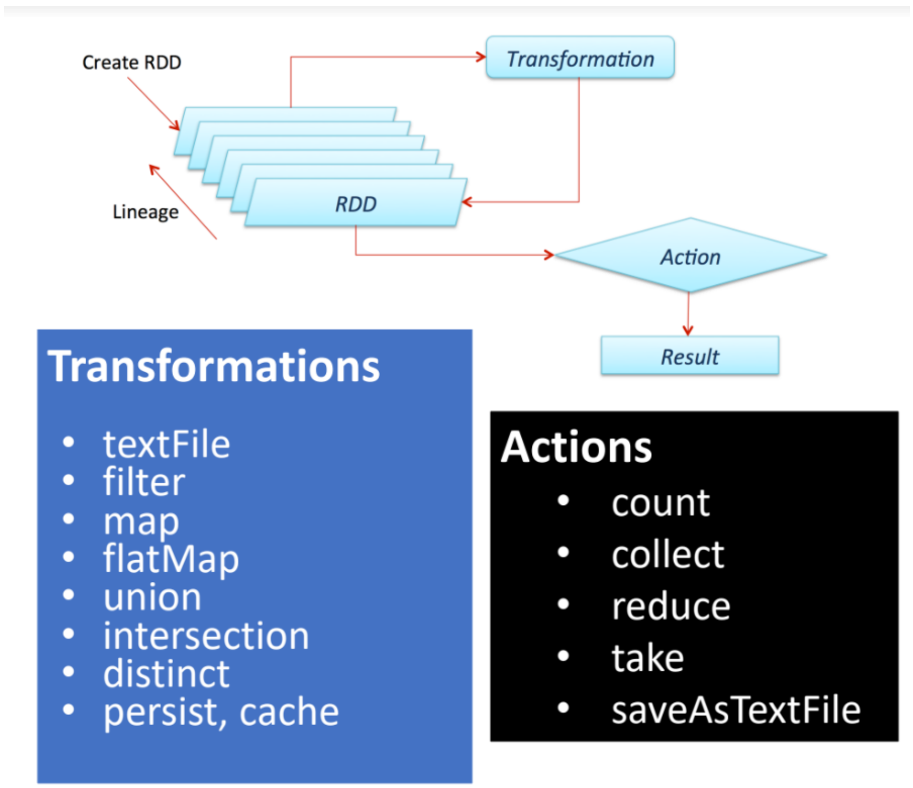
The `textFile()` method is used to read data from a text file on the local file system, HDFS, or any Hadoop-supported file system URI. This method returns an RDD of strings, with each element being a line from the text file.

Here's an example of using the `textFile()` method in Python with PySpark:

```
# Read a text file into an RDD
rdd = sc.textFile("file:///path/to/your/textfile.txt")

# Collect the results back to the driver program
lines = rdd.collect()
for line in lines:
    print(line)

# Save the RDD as a text file
rdd.saveAsTextFile("file:///path/to/output")
```

RDDs support two types of operations:

1. **Transformations**: These are operations that produce a new RDD from an existing RDD. Transformations are lazy, meaning they do not compute their results immediately. Instead, they build up a directed acyclic graph (DAG) representing the computation, which is executed only when an action is called.
2. **Actions**: These are operations that trigger the execution of the DAG and return a result to the driver program or write data to external storage.



Transformations:

1. Map:

- **Purpose**: The map transformation serves to apply a specified function to each element within the RDD, consequently generating a new RDD containing the transformed results.
- **Example**:

```
data = sc.parallelize(["value1,value2,value3", "value4,value5,value6",
...])
each_line_data = data.map(lambda line: line.split(","))
```

- **Explanation**: In this example, the RDD `data` is created using `sc.parallelize()` with a list of strings representing lines of values separated by commas. Then, the `map` transformation is utilized to split each line within the RDD `data` based on comma separation. The resultant RDD, `each_line_data`, comprises elements represented as lists, where each list contains values separated by commas.

Output:

```
each_line_data = [['value1', 'value2', 'value3'], ['value4', 'value5', 'value
6'], ...]
```

2. FlatMap:

- **Purpose**: The flatMap transformation is similar to map but differs in its output. It applies a function to each element in the RDD and returns a new RDD by flattening the results. This means that each element of the input RDD can produce zero or more elements in the output RDD.
- **Example**:

```
data = sc.parallelize(["value1,value2,value3", "value4,value5,value6",
...])
flat_mapped_data = data.flatMap(lambda line: line.split(","))
```

- **Explanation**: In this example, the RDD `data` is created using `sc.parallelize()` with a list of strings representing lines of values separated by commas. Then, the `flatMap` transformation is utilized to split each line within the RDD `data` based on comma separation and flatten the resulting list of lists into a single list. The resultant RDD, `flat_mapped_data`, comprises individual values rather than lists of values.

Output:

```
flat_mapped_data = ['value1', 'value2', 'value3', 'value4', 'value5', 'value
6', ...]
```

3. Filter:

- **Purpose**: The filter transformation is used to select elements from an RDD that satisfy a given predicate function, returning a new RDD containing only those elements.
- **Example**:

  ```
  data = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  filtered_data = data.filter(lambda x: x % 2 == 0)
  ```

- **Explanation**: In this example, the RDD `data` is created using `sc.parallelize()` with a list of integers. The `filter` transformation is then applied to select only the elements from `data` that satisfy the predicate function `x % 2 == 0`, which checks if the element is even. The resultant RDD, `filtered_data`, contains only the even numbers from the original RDD.

Output:

```
filtered_data = [2, 4, 6, 8, 10]
```

4. ReduceByKey:

- **Purpose**: The reduceByKey transformation is used specifically on key-value pair RDDs. It combines values with the same key using a specified associative function.
- **Example**:

  ```
  data = sc.parallelize([("a", 1), ("b", 2), ("a", 3), ("b", 4), ("c", 5)])
  reduced_data = data.reduceByKey(lambda x, y: x + y)
  ```

- **Explanation**: In this example, the RDD `data` is created using `sc.parallelize()` with a list of key-value pairs. The `reduceByKey` transformation is then applied to combine values with the same key. The lambda function `lambda x, y: x + y` is used to sum up the values associated with each key. The resultant RDD, `reduced_data`, contains unique keys with their associated aggregated values.

Output:

```
reduced_data = [("a", 4), ("b", 6), ("c", 5)]
```

```
# To Get Max Value
max_data = data.reduceByKey(lambda x, y: max(x, y))

# To Get Min Value
min_data = data.reduceByKey(lambda x, y: min(x, y))
```

5. SortBy:

- **Purpose**: The sortBy transformation is used to sort the elements of an RDD based on a specified key or a custom sorting function.
- **Example**:

```
data = sc.parallelize([("b", 2), ("a", 1), ("c", 3)])
sorted_data = data.sortBy(lambda x: x[0], ascending = True)
```

- **Explanation**: In this example, the RDD `data` is created using `sc.parallelize()` with a list of key-value pairs. The `sortBy` transformation is then applied to sort the elements of the RDD based on the key, which is accessed through the lambda function `lambda x: x[0]`. This sorts the RDD lexicographically based on the keys. The resultant RDD, `sorted_data`, contains elements sorted by their keys.

Output:

```
sorted_data = [("a", 1), ("b", 2), ("c", 3)]
```

6. Union:

- **Purpose**: Combines two RDDs into a single RDD containing all the elements from both RDDs.
- **Example**:

```
RDD3 = RDD1.union(RDD2)
```

7. Intersection:

- **Purpose**: `intersection` returns an RDD that contains the intersection of elements present in both input RDDs.
- **Example**:

```
common_elements_RDD = RDD1.intersection(RDD2)
```

8. Subtract:

- **Purpose:** The `subtract` operation in PySpark returns an RDD that contains elements present in the first RDD but not in the second RDD. It essentially subtracts the elements of the second RDD from the first RDD.
- **Example:**

```
unique_elements_RDD = RDD1.subtract(RDD2)
```

Here, `unique_elements_RDD` will contain elements that are present in `RDD1` but not in `RDD2`.

9. Distinct:

- **Purpose**: The `distinct` transformation removes duplicate elements from an RDD, returning a new RDD with only unique elements.
- **Example**:

```
distinct_elements_RDD = input_RDD.distinct()
```

- **Explanation**: Creates a new RDD `distinct_elements_RDD` containing only unique elements from the `input_RDD`. Duplicate elements are removed, leaving only one occurrence of each unique element.

10. MapValues:

- **Purpose**: The mapValues transformation is used to apply a function to the values of each key-value pair in an RDD without changing the keys.
- **Example**:

  ```
  data = sc.parallelize([("a", 1), ("b", 2), ("c", 3)])
  mapped_data = data.mapValues(lambda x: x * 2)
  ```

- **Explanation**: In this example, the RDD `data` is created using `sc.parallelize()` with a list of key-value pairs. The `mapValues` transformation is then applied to double the values of each key-value pair. The keys remain unchanged. The resultant RDD, `mapped_data`, contains key-value pairs with the values doubled.

Output:

```
mapped_data = [("a", 2), ("b", 4), ("c", 6)]
```

11. GroupByKey:

- **Purpose**: The groupByKey transformation is used to group together values associated with the same key in an RDD of key-value pairs.
- **Example**:

  ```
  data = sc.parallelize([("a", 1), ("b", 2), ("a", 3), ("b", 4), ("c", 5)])
  grouped_data = data.groupByKey()
  ```

- **Explanation**: In this example, the RDD `data` is created using `sc.parallelize()` with a list of key-value pairs. The `groupByKey` transformation is then applied to group together values associated with the same key. The resultant RDD, `grouped_data`, contains key-value pairs where each key is associated with an iterable of its corresponding values.

Output:

```
grouped_data = [("a", [1, 3]), ("b", [2, 4]), ("c", [5])]
```

12. Join Statements:

Join:

- **Purpose**: Combines elements from two RDDs based on matching keys.
- **Behavior**: Retains only the key-value pairs for which there are matching keys in both RDDs.
- **Example**:

```
rdd1 = sc.parallelize([(1, 'a'), (2, 'b')])
rdd2 = sc.parallelize([(1, 'x'), (3, 'y')])
joined_rdd = rdd1.join(rdd2)
```

- **Output**:

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

Left Join:

- **Purpose**: Combines elements from two RDDs, retaining all elements from the left RDD.
- **Behavior**: If a key in the left RDD has no match in the right RDD, the corresponding value in the result will be `None`.
- **Example**:

```
rdd1 = sc.parallelize([(1, 'a'), (2, 'b')])
rdd2 = sc.parallelize([(1, 'x'), (3, 'y')])
left_joined_rdd = rdd1.leftOuterJoin(rdd2)
```

- **Output**:

```
[(1, ('a', 'x')), (2, ('b', None))]
```

Right Join:

- **Purpose**: Combines elements from two RDDs, retaining all elements from the right RDD.
- **Behavior**: If a key in the right RDD has no match in the left RDD, the corresponding value in the result will be `None`.
- **Example**:

```
rdd1 = sc.parallelize([(1, 'a'), (2, 'b')])
rdd2 = sc.parallelize([(1, 'x'), (3, 'y')])
right_joined_rdd = rdd1.rightOuterJoin(rdd2)
```

- **Output**:

```
[(1, ('a', 'x')), (None, ('b', 'y'))]
```

13. FlatMapValues:

- **Purpose:** The `flatMapValues` transformation in PySpark is used to transform each value of a key-value pair RDD into zero or more output values, while retaining the original keys. It is similar to the `flatMap` transformation but operates specifically on the values of key-value pairs.

- **Syntax:**

```
new_rdd = rdd.flatMapValues(lambda value: iterable_function(value))
```

Actions:

1. take:

- **Purpose**: Retrieves the first `n` elements from an RDD and returns them as a list.
- **Example**:

```
taken_elements = data.take(5)
```

2. count:

- **Purpose**: The `count` action returns the total number of elements in an RDD.
- **Example**:

```
total_count = input_RDD.count()
```

- **Explanation**: Calculates and returns the total number of elements in the `input_RDD`. This action is useful for determining the size of an RDD or for counting the number of occurrences of certain elements.

3. countByValue:

The `countByValue` action in Apache Spark counts the occurrences of each unique value in an RDD and returns the result as a dictionary where the keys are the distinct values and the values are their respective counts.

- **Example**:

```
data = sc.parallelize([1, 2, 3, 1, 2, 1, 3, 4, 5])
value_counts = data.countByValue()
```

- **Explanation**: In this example, an RDD named `data` is created using `sc.parallelize()` with a list of integers. The `countByValue` action is then applied to count the occurrences of each unique value in the RDD. The result is stored in the `value_counts` variable.
- **Output**:

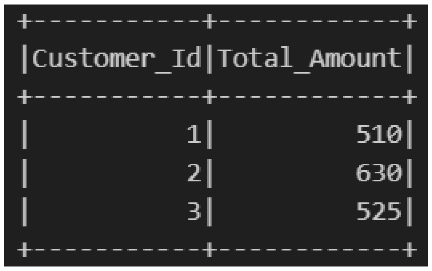
```
value_counts = {1: 3, 2: 2, 3: 2, 4: 1, 5: 1}
```

Here's an example of SPARK CODE in Python using PySpark

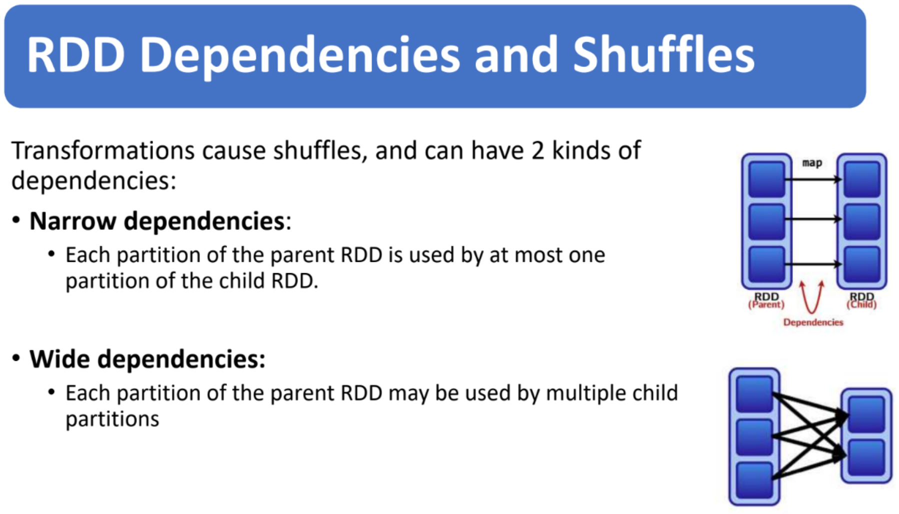
You have a dataset transaction.csv
customer_id, transaction_amount, transaction_date

1,100,2023-01-05
2,150,2023-01-10
1,200,2023-02-15
3,75,2023-03-20
2,300,2023-04-25
1,120,2023-05-30
3,250,2023-06-05
2,180,2023-07-10
1,90,2023-08-15
3,200,2023-09-20
Find the total transaction amount for each customer for the year 2023.

```
import pyspark
from pyspark import SparkContext as BabySharkContext, SparkConf as BabySharkConf
from pyspark.sql import SparkSession as BabySharkSession

conf = BabySharkConf().setAppName("TASK 1").setMaster("local")

sc = BabySharkContext.getOrCreate(conf = conf)
sharky = BabySharkSession(sc)

Data = sc.textFile("Task1_Input.csv")
each_line_Data = Data.map(lambda line: line.split(","))
Data_2023 = each_line_Data.filter(lambda row: "2023" in row[2])
Mapped_Data = Data_2023.map(lambda row: (int(row[0]), int(row[1])))
Total_Transactions_2023 = Mapped_Data.reduceByKey(lambda x, y: x + y)

DataFrame = sharky.createDataFrame(Total_Transactions_2023, ["Customer_Id", "Tot
DataFrame.show()

sc.stop()

OR

import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder \
.master('local[*]') \
.appName('Basics') \
.getOrCreate()
sc = spark.sparkContext
sc.stop
```



```
+-----------+------------+
|Customer_Id|Total_Amount|
+-----------+------------+
|          1|         510|
|          2|         630|
|          3|         525|
+-----------+------------+
```



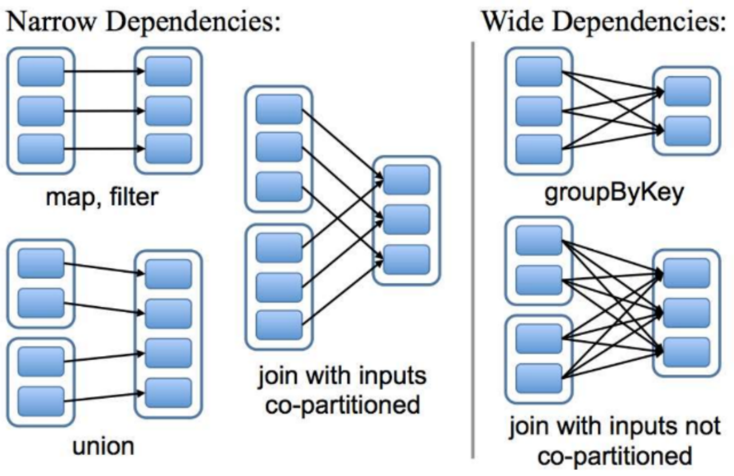
RDD Dependencies and Shuffles

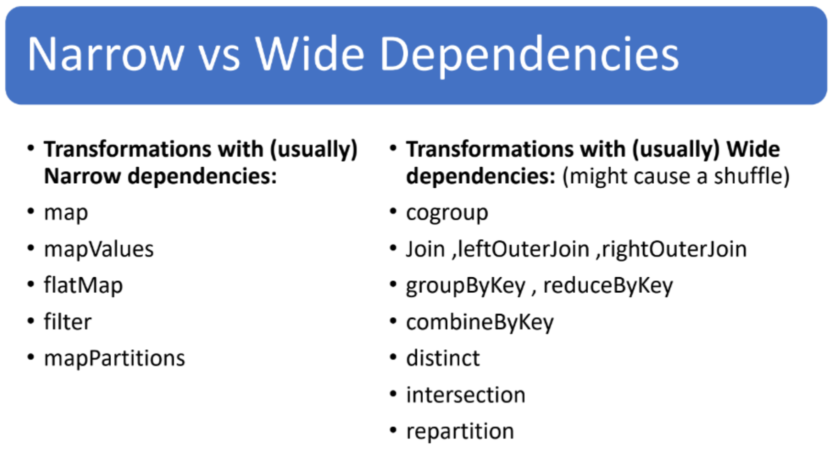
Transformations cause shuffles, and can have 2 kinds of dependencies:

- **Narrow dependencies**:
  - Each partition of the parent RDD is used by at most one partition of the child RDD.

- **Wide dependencies:**
  - Each partition of the parent RDD may be used by multiple child partitions

map
RDD (Parent)
RDD (Child)
Dependencies


Narrow Dependencies:
Wide Dependencies:
map, filter
union
join with inputs co-partitioned
groupByKey
join with inputs not co-partitioned



Narrow vs Wide Dependencies

- **Transformations with (usually) Narrow dependencies:**
- map
- mapValues
- flatMap
- filter
- mapPartitions

- **Transformations with (usually) Wide dependencies:** (might cause a shuffle)
- cogroup
- Join ,leftOuterJoin ,rightOuterJoin
- groupByKey , reduceByKey
- combineByKey
- distinct
- intersection
- repartition

Persist Caching in PySpark

Persisting and **caching** are optimization techniques in PySpark to improve the performance of your applications by storing intermediate results. This can be particularly useful when you have a dataset that is reused multiple times in different stages of your computations.

- **Cache**: The `cache()` method stores the RDD (Resilient Distributed Dataset) in memory. This is useful for repeated operations on the same dataset. By default, `cache()` uses the `MEMORY_ONLY` storage level.
- **Persist**: The `persist()` method is more flexible compared to `cache()`, allowing you to specify different storage levels such as `MEMORY_ONLY`, `MEMORY_AND_DISK`, `DISK_ONLY`, etc.

Example usage:

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.persist()  # Can also use rdd.cache()

result1 = rdd.map(lambda x: x * x).collect()
result2 = rdd.filter(lambda x: x % 2 == 0).collect()
```

In this example, the RDD `rdd` is persisted in memory, so its data does not need to be recomputed for the subsequent actions (`map` and `filter`).

Broadcast Variables

Broadcast variables are used to efficiently distribute small read-only data values to all worker nodes. Instead of shipping a copy of the variable with each task, Spark distributes it once using efficient broadcast algorithms to reduce communication costs.

Example usage:

```
broadcastVar = sc.broadcast([1, 2, 3, 4, 5])
broadcastVar.value
```

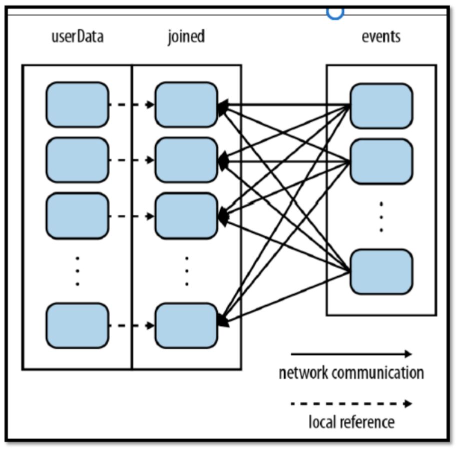
In this example, the list `[1, 2, 3, 4, 5]` is broadcasted to all nodes, and each task can access this list without needing to send it over the network multiple times.

Data partitioning:

Data partitioning in Spark involves dividing data across partitions to optimize performance and reduce network traffic. Spark automatically infers default values based on cluster size, but users can specify the number of partitions for operations like aggregations or grouping.

To split work, Spark employs functions like `reduceByKey` or `groupByKey`, where users can specify the number of partitions. The `repartition()` function shuffles data to create new partitions, while `coalesce()` combines partitions, but it's less expensive.

Partitioning minimizes network traffic, especially in key-oriented operations like joins. It's most useful when datasets are reused multiple times. For example, partitioning user data can optimize operations like joining with event data.



Partitioned userData joined with Events

Here's how you can use `partitionBy()` to partition user data:

```
# Assuming 'rdd' is a Pair RDD with (key, value) pairs
# Partitioning by a custom key
partitioned_rdd = rdd.partitionBy(10)  # Partitioning into 10 partitions

# Assuming 'rdd' is an RDD that needs to be repartitioned
# Repartitioning to 10 partitions
repartitioned_rdd = rdd.repartition(10)
```

This transformation returns a new RDD, so it's important to save the result as `userData`.

Spark automatically sets partitioners for operations like `cogroup()`, `groupByKey()`, or `reduceByKey()`, optimizing performance. However, transformations like `map()` may lose partitioning information.

Custom partitioning functions can also be defined:

```
def custom_partitioner(key):
    # Custom logic to determine the partition for each key
    return hash(key) % 10  # Modulo 10 partitioning

partitioned_rdd = rdd.partitionBy(10, custom_partitioner)
```

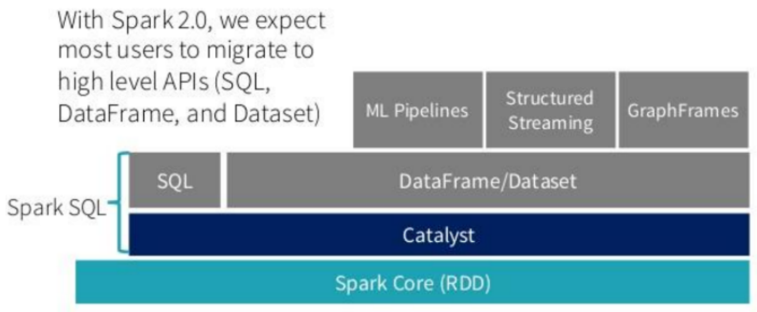
This function, `custom_partitioner`, takes a record as input and applies custom logic to determine which partition the record should belong to. In this case, it calculates the hash value of the record and performs modulo division by 10, ensuring that records are distributed across 10 partitions.

SQL with RDD in PySpark

RDDs (Resilient Distributed Datasets) are not good at handling structured data and lack built-in optimization. The solution to this is DataFrames.

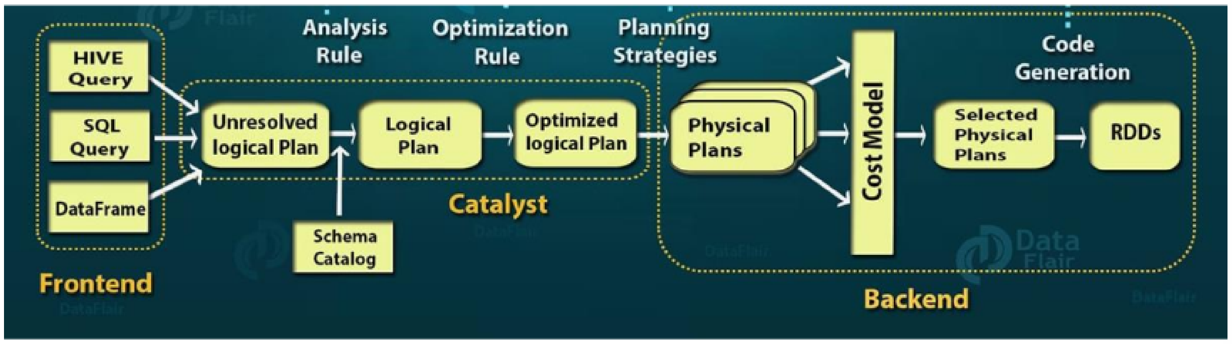
DataFrames are like tables in a database or in R/Python. They organize data into named columns and have a structure with an optimization engine called Catalyst.

Compared to RDDs, DataFrames offer memory management and optimized execution plans, making them more efficient. Both RDDs and DataFrames share features like immutability, in-memory storage, resilience, and distribution.


With Spark 2.0, we expect most users to migrate to high level APIs (SQL, DataFrame, and Dataset)
ML Pipelines
Structured Streaming
GraphFrames
Spark SQL
SQL
DataFrame/Dataset
Catalyst
Spark Core (RDD)

DataFrames provide a higher-level abstraction, allowing users to impose structure on distributed data collections.

DataFrame uses **Catalyst tree transformation** in four phases:



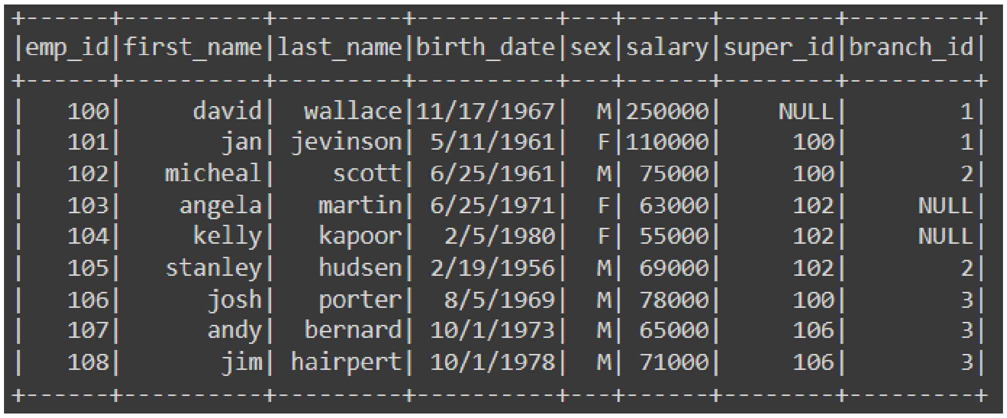
Spark SQL provides a structured data processing module within Apache Spark, offering DataFrames for easy manipulation and querying.

1. Setting up SparkSession:

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession

conf = SparkConf().setAppName("SQL Queries").setMaster("local")
sc = SparkContext.getOrCreate(conf=conf)
spark = SparkSession(sc)
```

2. Reading CSV File:

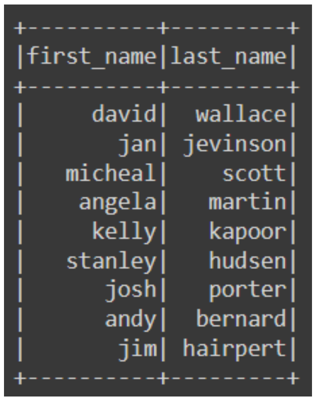
```
data = spark.read.csv("EmployeeData.csv", header=True, inferSchema=True)
data.show()
```



```
+------+----------+---------+----------+---+------+--------+---------+
|emp_id|first_name|last_name|birth_date|sex|salary|super_id|branch_id|
+------+----------+---------+----------+---+------+--------+---------+
|   100|     david|  wallace|11/17/1967|  M|250000|    NULL|        1|
|   101|       jan| jevinson| 5/11/1961|  F|110000|     100|        1|
|   102|   micheal|    scott| 6/25/1961|  M| 75000|     100|        2|
|   103|    angela|   martin| 6/25/1971|  F| 63000|     102|     NULL|
|   104|     kelly|   kapoor|  2/5/1980|  F| 55000|     102|     NULL|
|   105|   stanley|   hudsen| 2/19/1956|  M| 69000|     102|        2|
|   106|      josh|   porter|  8/5/1969|  M| 78000|     100|        3|
|   107|      andy|  bernard| 10/1/1973|  M| 65000|     106|        3|
|   108|       jim| hairpert| 10/1/1978|  M| 71000|     106|        3|
+------+----------+---------+----------+---+------+--------+---------+
```

3. Selecting Columns:

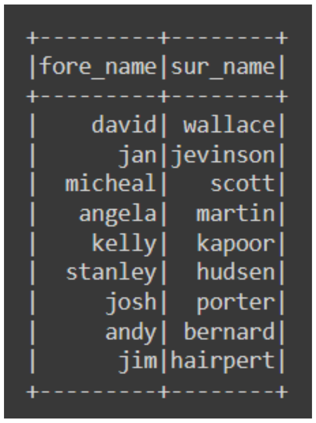
```
# Find first and last name of all Employees
RDD1 = data.select(data.first_name, data.last_name)
RDD1.show()
```



```
+----------+---------+
|first_name|last_name|
+----------+---------+
|     david|  wallace|
|       jan| jevinson|
|   micheal|    scott|
|    angela|   martin|
|     kelly|   kapoor|
|   stanley|   hudsen|
|      josh|   porter|
|      andy|  bernard|
|       jim| hairpert|
+----------+---------+
```

4. Renaming Columns:

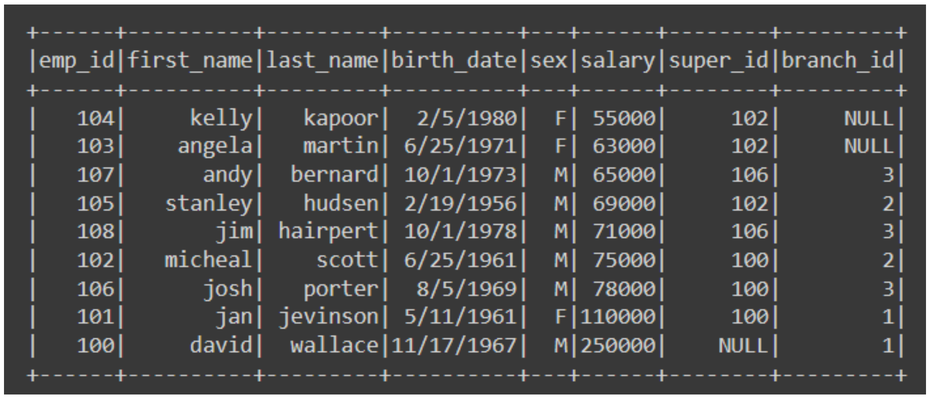
```
# Find fore and sur name of all Employees
RDD2 = data.select(data.first_name.alias("fore_name"), data.last_name.alias
("sur_name"))
RDD2.show()
```



```
+---------+--------+
|fore_name|sur_name|
+---------+--------+
|    david| wallace|
|      jan|jevinson|
|  micheal|   scott|
|   angela|  martin|
|    kelly|  kapoor|
|  stanley|  hudsen|
|     josh|  porter|
|     andy| bernard|
|      jim|hairpert|
+---------+--------+
```

5. Ordering Data:

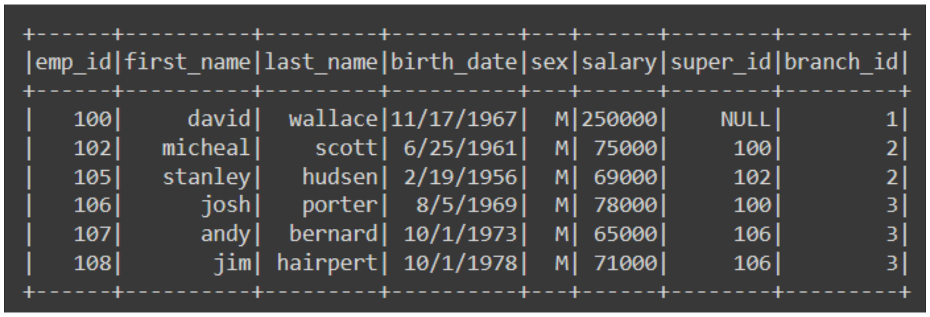
```
# Find all Employees ordered by salary (ascending order)
RDD3 = data.orderBy(data.salary, ascending=True)
RDD3.show()
```



```
+------+----------+---------+----------+---+------+--------+---------+
|emp_id|first_name|last_name|birth_date|sex|salary|super_id|branch_id|
+------+----------+---------+----------+---+------+--------+---------+
|   104|     kelly|   kapoor|  2/5/1980|  F| 55000|     102|     NULL|
|   103|    angela|   martin| 6/25/1971|  F| 63000|     102|     NULL|
|   107|      andy|  bernard| 10/1/1973|  M| 65000|     106|        3|
|   105|   stanley|   hudsen| 2/19/1956|  M| 69000|     102|        2|
|   108|       jim| hairpert| 10/1/1978|  M| 71000|     106|        3|
|   102|   micheal|    scott| 6/25/1961|  M| 75000|     100|        2|
|   106|      josh|   porter|  8/5/1969|  M| 78000|     100|        3|
|   101|       jan| jevinson| 5/11/1961|  F|110000|     100|        1|
|   100|     david|  wallace|11/17/1967|  M|250000|    NULL|        1|
+------+----------+---------+----------+---+------+--------+---------+
```

6. Filtering Data:

```
# Find all male Employees
RDD4 = data.filter(data.sex == 'M')
RDD4.show()
```



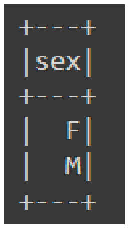
```
+------+----------+---------+----------+---+------+--------+---------+
|emp_id|first_name|last_name|birth_date|sex|salary|super_id|branch_id|
+------+----------+---------+----------+---+------+--------+---------+
|   100|     david|  wallace|11/17/1967|  M|250000|    NULL|        1|
|   102|   micheal|    scott| 6/25/1961|  M| 75000|     100|        2|
|   105|   stanley|   hudsen| 2/19/1956|  M| 69000|     102|        2|
|   106|      josh|   porter|  8/5/1969|  M| 78000|     100|        3|
|   107|      andy|  bernard| 10/1/1973|  M| 65000|     106|        3|
|   108|       jim| hairpert| 10/1/1978|  M| 71000|     106|        3|
+------+----------+---------+----------+---+------+--------+---------+
```

7. Combining Filters:

```
# Find all Employees form Branch 2 and Sex F
RDD5 = data.filter((data.branch_id == 2) & (data.sex == 'F'))
RDD5.show()
```

8. Finding Distinct Values:

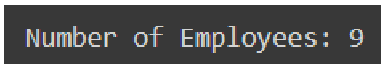
```
# Find out all distinct Gender
RDD6 = data.select(data.sex).distinct()
RDD6.show()
```



```
+---+
|sex|
+---+
|  F|
|  M|
+---+
```

9. Counting Rows:

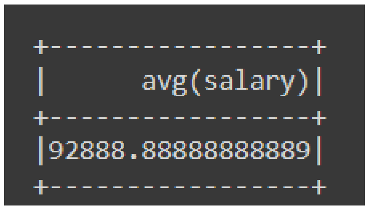
```
# Find number of Employee
RDD7 = data.count()
print("Number of Employees:", RDD7)
```



```
Number of Employees: 9
```

10. Aggregating Data - Average:

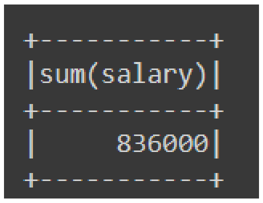
```
# Find average of all Employee salaries
from pyspark.sql.functions import avg
RDD8 = data.select(avg(data.salary))
RDD8.show()
```



```
+-----------------+
|      avg(salary)|
+-----------------+
|92888.88888888889|
+-----------------+
```

11. Aggregating Data - Sum:

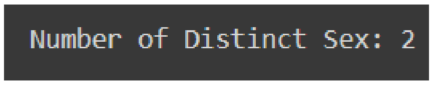
```
# Find sum of all Employee salaries
from pyspark.sql.functions import sum
RDD9 = data.select(sum(data.salary))
RDD9.show()
```



```
+-----------+
|sum(salary)|
+-----------+
|     836000|
+-----------+
```

12. Counting Distinct Values:

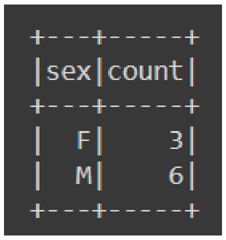
```
# Find number of distinct sex in employee table
RDD10 = data.select(data.sex).distinct().count()
print("Number of Distinct Sex:", RDD10)
```



```
Number of Distinct Sex: 2
```

13. Grouping Data:

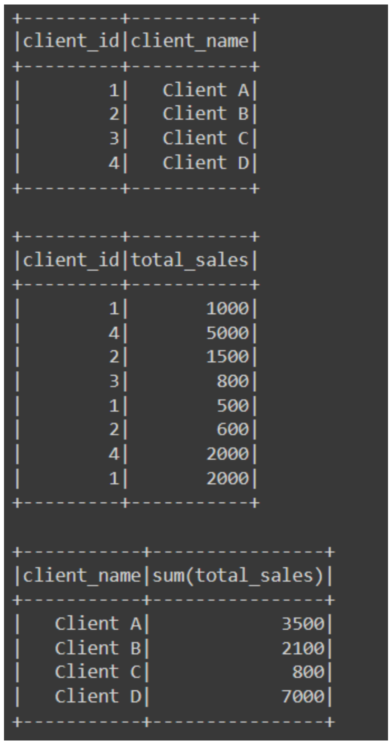
```
# Find how many sex in Employee Table
RDD11 = data.groupBy(data.sex).count()
RDD11.show()
```



```
+---+-----+
|sex|count|
+---+-----+
|  F|    3|
|  M|    6|
+---+-----+
```

14. Joining Data:

```
# Find out total sales greater than 560000 of each with Client Name
from pyspark.sql.functions import sum

Data1 = spark.read.csv("ClientsData.csv", header = True, inferSchema = True)
Data2 = spark.read.csv("WorksOnData.csv", header = True, inferSchema = True)

rdd1 = Data1.join(Data2, Data1.client_id == Data2.client_id).select(Data1.cli
ent_name, Data2.total_sales)
rdd2 = rdd1.groupBy(Data1.client_name)
rdd3 = rdd2.agg(sum("total_sales").alias("total_sales"))
rdd4 = rdd3.filter(rdd3.total_sales > 56000)
rdd4.show()

# avg("total_sales")
# min("total_sales")
# max("total_sales")
# count()

RDD12.show()
```



```
+---------+-----------+
|client_id|client_name|
+---------+-----------+
|        1|   Client A|
|        2|   Client B|
|        3|   Client C|
|        4|   Client D|
+---------+-----------+

+---------+-----------+
|client_id|total_sales|
+---------+-----------+
|        1|       1000|
|        4|       5000|
|        2|       1500|
|        3|        800|
|        1|        500|
|        2|        600|
|        4|       2000|
|        1|       2000|
+---------+-----------+

+-----------+----------------+
|client_name|sum(total_sales)|
+-----------+----------------+
|   Client A|            3500|
|   Client B|            2100|
|   Client C|             800|
|   Client D|            7000|
+-----------+----------------+
```

15. Stopping SparkContext:

```
sc.stop()
```

Here are some other important built-in functions:

1. `between`:
   - Syntax: `col("column_name").between(lower_bound, upper_bound)`
   - Used to filter rows where a column's value falls within a specified range.
   - Example: `filtered_df = products_df.filter((col("Price").between(10, 50)))`
2. `isin`:
   - Syntax: `df.filter(df["column_name"].isin(list_of_values))`
   - Filters rows where a column's value is present in a list of specified values.
   - Example: `filtered_df = orders_df.filter(orders_df["CustomerID"].isin(customer_ids))`
3. `~isin`:
   - Syntax: `df.filter(~df["column_name"].isin(list_of_values))`
   - Filters rows where a column's value is not present in a list of specified values.
   - Example: `filtered_df = orders_df.filter(~orders_df["CustomerID"].isin(customer_ids_to_exclude))`
4. `isnull`:
   - Syntax: `df.filter(df["column_name"].isNull())`

- Filters rows where a column's value is `NULL` or `None`.
- Example: `filtered_df = employees_df.filter(employees_df["middle_name"].isNull())`

5. **`isnotnull`**:
   - Syntax: `df.filter(df["column_name"].isNotNull())`
   - Filters rows where a column's value is not `NULL` or `None`.
   - Example: `filtered_df = employees_df.filter(employees_df["middle_name"].isNotNull())`
6. **`join`**:
   - Syntax: `df1.join(df2, df1["join_column"] == df2["join_column"], "join_type")`
   - Performs a join operation between two DataFrames based on a specified column.
   - Example: `joined_df = employees_df.join(departments_df, employees_df["department_id"] == departments_df["department_id"], "left")`
7. **`leftjoin`**, **`rightjoin`**:
   - Syntax: `df1.join(df2, df1["join_column"] == df2["join_column"], "left")`
   - Performs a left or right join operation between two DataFrames based on a specified column.
   - Example: `joined_df = employees_df.join(departments_df, employees_df["department_id"] == departments_df["department_id"], "left")`

SQL Queries in PySpark:

SQL queries in PySpark leverage the structured data processing capabilities of Spark, enabling users to interact with large-scale datasets using familiar SQL syntax.

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession

conf = SparkConf().setAppName("SQL Queries").setMaster("local")

sc = SparkContext.getOrCreate(conf = conf)
spark = SparkSession(sc)

df = spark.read.csv("EmployeeData.csv", header = True, inferSchema = True)
df.createOrReplaceTempView("EmployeeData")

# Find all Employees
sql_cmd = "SELECT * FROM EmployeeData"
df_result = spark.sql(sql_cmd)
df_result.show()

# Find first and last name of all Employees
sql_cmd = "SELECT first_name, last_name FROM EmployeeData"
```

```
df_result = spark.sql(sql_cmd)
df_result.show()

# Find fore and sur name of all Employees
sql_cmd = "SELECT first_name AS fore_name, last_name AS sur_name FROM EmployeeDa
df_result = spark.sql(sql_cmd)
df_result.show()

# Find all Employees ordered by salary (ascending order)
sql_cmd = "SELECT * FROM EmployeeData ORDER BY salary ASC"
df_result = spark.sql(sql_cmd)
df_result.show()

# Find all male Employees
sql_cmd = "SELECT * FROM EmployeeData WHERE sex ='M'"
df_result = spark.sql(sql_cmd)
df_result.show()

# Find average of all Employee salaries
sql_cmd = "SELECT AVG(salary) FROM EmployeeData"
df_result = spark.sql(sql_cmd)
df_result.show()

df1 = spark.read.csv("ClientsData.csv", header = True, inferSchema = True)
df1.createOrReplaceTempView("ClientsData")
df2 = spark.read.csv("WorksOnData.csv", header = True, inferSchema = True)
df2.createOrReplaceTempView("WorksOnData")

# Find out total sales of each with Client Name
sql_cmd = '''SELECT ClientsData.client_name, SUM(WorksOnData.total_sales) AS Sal
             FROM ClientsData
             JOIN WorksOnData ON ClientsData.client_id = WorksOnData.client_id
             GROUP BY ClientsData.client_name'''
df_result = spark.sql(sql_cmd)
df_result.show()

sc.stop()
```