SPARK

PySpark Tutorial

- PySpark Tutorial For Beginners
- PySpark – Features
- PySpark – Advantages
- PySpark – Modules & Packages
- PySpark – Cluster Managers
- PySpark – Install on Windows
- PySpark – Install on Mac
- PySpark – Web/Application UI
- PySpark – SparkSession
- PySpark – SparkContext
- PySpark – RDD
- PySpark – Parallelize
- PySpark – repartition() vs coalesce()
- PySpark – Broadcast Variables
- PySpark – Accumulator

PySpark DataFrame

- PySpark – Create a DataFrame
- PySpark – Create an empty DataFrame
- PySpark – Convert RDD to DataFrame
- PySpark – Convert DataFrame to Pandas
- PySpark – show()
- PySpark – StructType & StructField
- PySpark – Row Class
- PySpark – Column Class
- PySpark – select()
- PySpark – collect()
- PySpark – withColumn()
- PySpark – withColumnRenamed()
- PySpark – where() & filter()

- PySpark – drop() & dropDuplicates()
- PySpark – orderBy() and sort()
- PySpark – groupBy()
- PySpark – join()
- PySpark – union() & unionAll()
- PySpark – unionByName()
- PySpark – UDF (User Defined Function)
- PySpark – map()
- PySpark – flatMap()
- pyspark – foreach()
- PySpark – sample() vs sampleBy()
- PySpark – fillna() & fill()
- PySpark – pivot() (Row to Column)
- PySpark – partitionBy()
- PySpark – ArrayType Column (Array)
- PySpark – MapType (Map/Dict)

PySpark SQL Functions

- PySpark – Aggregate Functions
- PySpark – Window Functions
- PySpark – Date and Timestamp Functions
- PySpark – JSON Functions

PySpark Datasources

- PySpark – Read & Write CSV File
- PySpark – Read & Write Parquet File
- PySpark – Read & Write JSON file

PySpark Built-In Functions

- PySpark – when()
- PySpark – expr()
- PySpark – lit()
- PySpark – split()
- PySpark – concat_ws()

- Pyspark – substring()
- PySpark – translate()
- PySpark – regexp_replace()
- PySpark – overlay()
- PySpark – to_timestamp()
- PySpark – to_date()
- PySpark – date_format()
- PySpark – datediff()
- PySpark – months_between()
- PySpark – explode()
- PySpark – array_contains()
- PySpark – array()
- PySpark – collect_list()
- PySpark – collect_set()
- PySpark – create_map()
- PySpark – map_keys()
- PySpark – map_values()
- PySpark – struct()
- PySpark – countDistinct()
- PySpark – sum(), avg()
- PySpark – row_number()
- PySpark – rank()
- PySpark – dense_rank()
- PySpark – percent_rank()
- PySpark – typedLit()
- PySpark – from_json()
- PySpark – to_json()
- PySpark – json_tuple()
- PySpark – get_json_object()
- PySpark – schema_of_json()

Apache Spark™ examples

These examples give a quick overview of the Spark API. Spark is built on the concept of *distributed datasets*, which contain arbitrary Java or Python objects. You create a dataset from external data, then apply parallel operations to it. The building block of the Spark API is its RDD API. In the RDD API, there are two types of operations: *transformations*, which define a new dataset based on previous ones, and *actions*, which kick off a job to execute on a cluster. On top of Spark's RDD API, high level APIs are provided, e.g. DataFrame API and Machine Learning API. These high level APIs provide a concise way to conduct certain data operations. In this page, we will show examples using RDD API as well as examples using high level APIs.

RDD API examples

Word count

In this example, we use a few transformations to build a dataset of (String, Int) pairs called `counts` and then save it to a file.

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

Pi estimation

Spark can also be used for compute-intensive tasks. This code estimates π by "throwing darts" at a circle. We pick random points in the unit square ((0, 0) to (1,1)) and see how many fall in the unit circle. The fraction should be $\pi / 4$, so we use this to get our estimate.

- Python
- Scala
- Java

```
def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y < 1

count = sc.parallelize(range(0, NUM_SAMPLES)) \
             .filter(inside).count()
```

```
print("Pi is roughly %f" % (4.0 * count / NUM_SAMPLES))
```

DataFrame API examples

In Spark, a DataFrame is a distributed collection of data organized into named columns. Users can use DataFrame API to perform various relational operations on both external data sources and Spark's built-in distributed collections without providing specific procedures for processing data. Also, programs based on DataFrame API will be automatically optimized by Spark's built-in optimizer, Catalyst.

Text search

In this example, we search through the error messages in a log file.

- Python
- Scala
- Java

```
textFile = sc.textFile("hdfs://...")


# Creates a DataFrame having a single column named "line"

df = textFile.map(lambda r: Row(r)).toDF(["line"])

errors = df.filter(col("line").like("%ERROR%"))

# Counts all the errors

errors.count()

# Counts errors mentioning MySQL

errors.filter(col("line").like("%MySQL%")).count()

# Fetches the MySQL errors as an array of strings

errors.filter(col("line").like("%MySQL%")).collect()
```

Simple data operations

In this example, we read a table stored in a database and calculate the number of people for every age. Finally, we save the calculated result to S3 in the format of JSON. A simple MySQL table "people" is used in the example and this table has two columns, "name" and "age".

- Python
- Scala

- Java

```
# Creates a DataFrame based on a table named "people"
# stored in a MySQL database.
url = \
  "jdbc:mysql://yourIP:yourPort/test?user=yourUsername;password=yourPassword"
df = sqlContext \
  .read \
  .format("jdbc") \
  .option("url", url) \
  .option("dbtable", "people") \
  .load()

# Looks the schema of this DataFrame.
df.printSchema()

# Counts people by age
countsByAge = df.groupBy("age").count()
countsByAge.show()

# Saves countsByAge to S3 in the JSON format.
countsByAge.write.format("json").save("s3a://...")
```

Machine learning example

MLlib, Spark's Machine Learning (ML) library, provides many distributed ML algorithms. These algorithms cover tasks such as feature extraction, classification, regression, clustering, recommendation, and more. MLlib also provides tools such as ML Pipelines for building workflows, CrossValidator for tuning parameters, and model persistence for saving and loading models.

Prediction with logistic regression

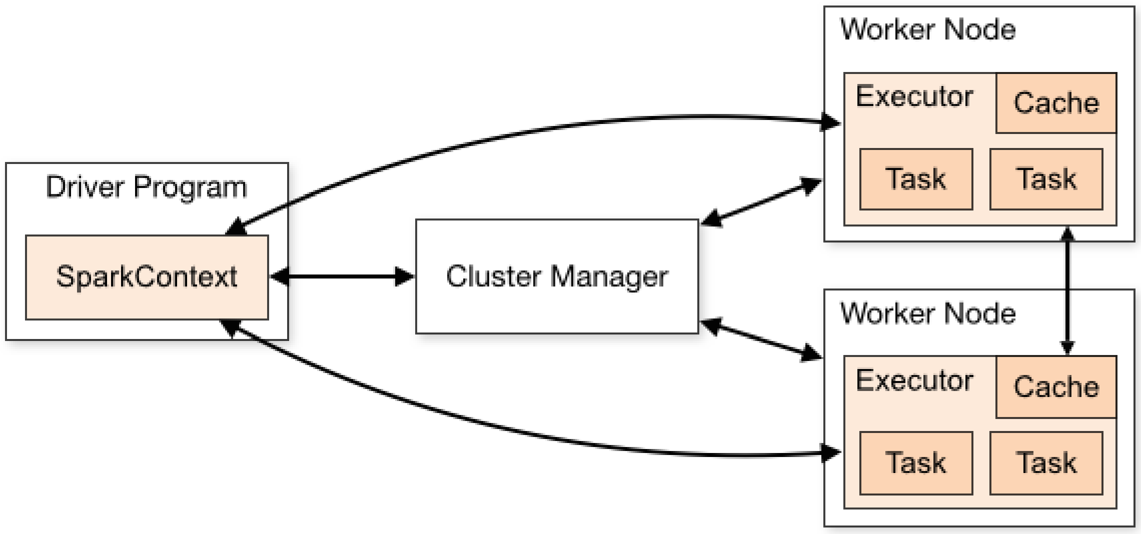
In this example, we take a dataset of labels and feature vectors. We learn to predict the labels from feature vectors using the Logistic Regression algorithm.

- Python
- Scala
- Java

```python
# Every record of this DataFrame contains the label and
# features represented by a vector.
df = sqlContext.createDataFrame(data, ["label", "features"])

# Set parameters for the algorithm.
# Here, we limit the number of iterations to 10.
lr = LogisticRegression(maxIter=10)

# Fit the model to the data.
model = lr.fit(df)

# Given a dataset, predict each point's label, and show the results.
model.transform(df).show()
```

PySpark Architecture

Apache Spark works in a master-slave architecture where the master is called “Driver” and slaves are called “Workers”. When you run a Spark application, Spark Driver creates a context that is an entry point to your application, and all operations (transformations and actions) are executed on worker nodes, and the resources are managed by Cluster Manager.



source: https://spark.apache.org/

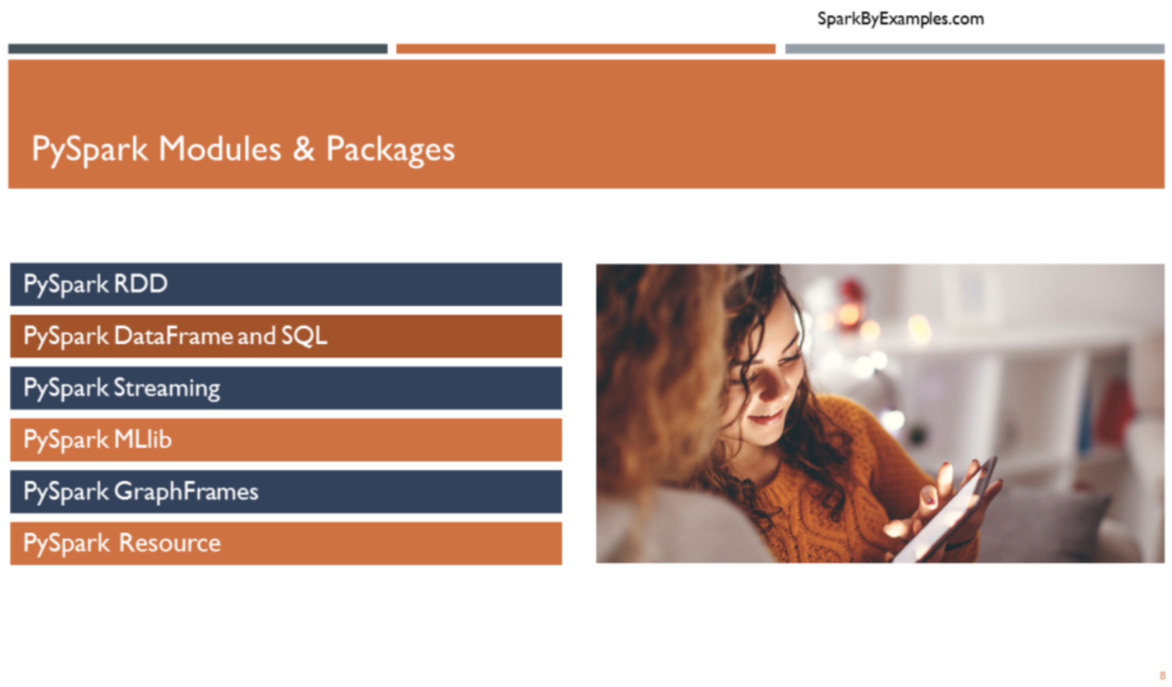
Cluster Manager Types

As of writing this Spark with Python (PySpark) tutorial, Spark supports below cluster managers:

- Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos – Mesons is a Cluster manager that can also run Hadoop MapReduce and PySpark applications.
- Hadoop YARN – the resource manager in Hadoop 2. This is mostly used, cluster manager.
- Kubernetes – an open-source system for automating deployment, scaling, and management of containerized applications.

local – which is not really a cluster manager but still I wanted to mention as we use "local" for `master()` in order to run Spark on your laptop/computer.

PySpark Modules & Packages


SparkByExamples.com

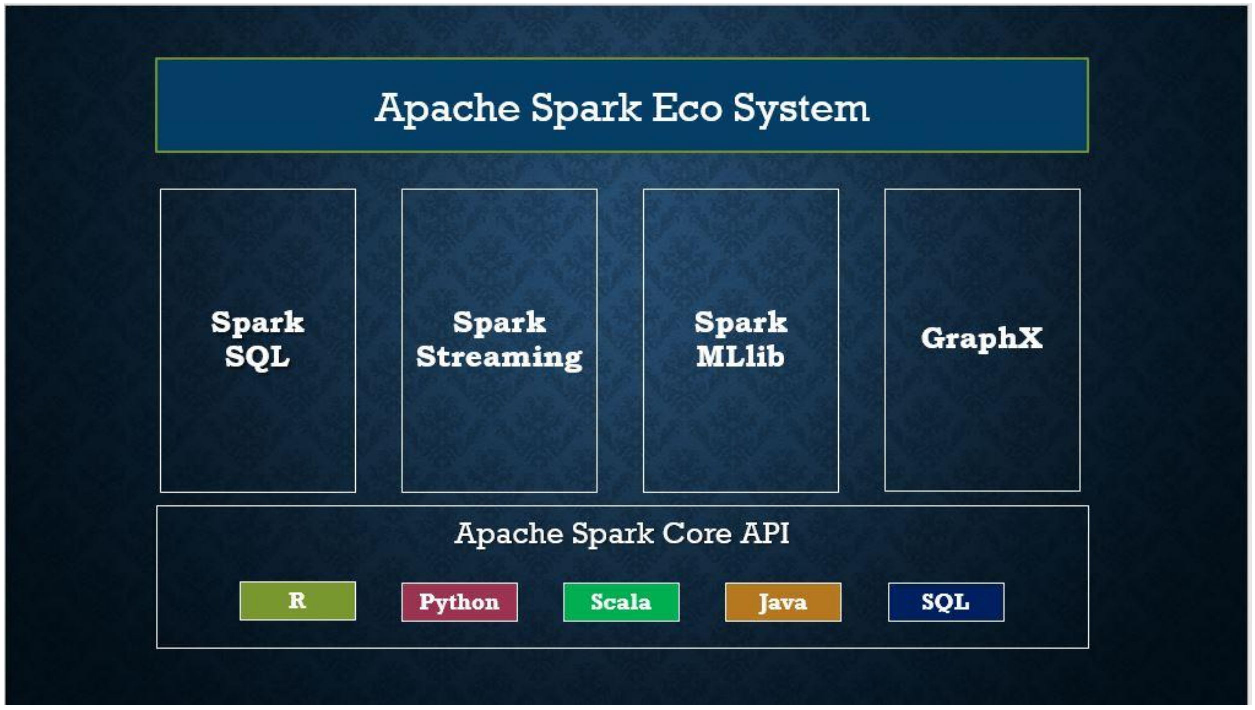
PySpark Modules & Packages

- PySpark RDD
- PySpark DataFrame and SQL
- PySpark Streaming
- PySpark MLlib
- PySpark GraphFrames
- PySpark Resource

8

Modules & packages

- PySpark RDD (pyspark.RDD)
- PySpark DataFrame and SQL (pyspark.sql)
- PySpark Streaming (pyspark.streaming)
- PySpark MLib (pyspark.ml, pyspark.mllib)
- PySpark GraphFrames (GraphFrames)
- PySpark Resource (pyspark.resource) It’s new in PySpark 3.0

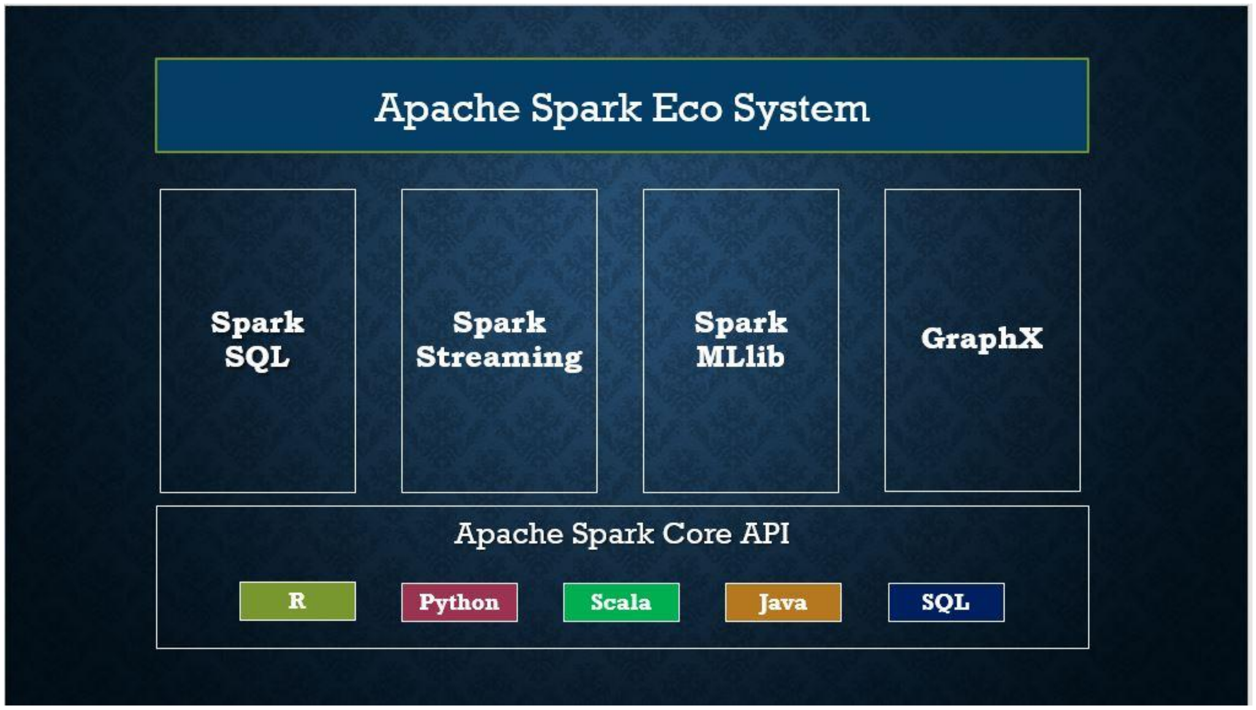


Besides these, if you wanted to use third-party libraries, you can find them at https://spark-packages.org/ . This page is kind of a repository of all Spark third-party libraries.

PySpark Installation

Spark Modules

- Spark Core
- Spark SQL
- Spark Streaming
- Spark MLlib
- Spark GraphX



Spark Modules

Spark Core

In this section of the Apache Spark Tutorial, you will learn different concepts of the Spark Core library with examples in Scala code. Spark Core is the main base library of the Spark which provides the abstraction of how distributed task dispatching, scheduling, basic I/O functionalities and etc.

Before getting your hands dirty on Spark programming, have your Development Environment Setup to run Spark Examples using IntelliJ IDEA

SparkSession

SparkSession introduced in version 2.0, It is an entry point to underlying Spark functionality in order to programmatically use Spark RDD, DataFrame and Dataset. It's object `spark` is default available in spark-shell.

Creating a SparkSession instance would be the first statement you would write to program with RDD, DataFrame and Dataset. SparkSession will be created using `SparkSession.builder()` builder pattern.

```
import org.apache.spark.sql.SparkSession
val spark:SparkSession = SparkSession.builder()
      .master("local[1]")
      .appName("SparkByExamples.com")
      .getOrCreate()
```

Copy

Spark Context

SparkContext is available since Spark 1.x (JavaSparkContext for Java) and is used to be an entry point to Spark and PySpark before introducing SparkSession in 2.0. Creating SparkContext was the first step to the program with RDD and to connect to Spark Cluster. It's object `sc` by default available in `spark-shell`.

Since Spark 2.x version, When you create SparkSession, SparkContext object is by default create and it can be accessed using `spark.sparkContext`

Note that you can create just one SparkContext per JVM but can create many SparkSession objects.

RDD Spark Tutorial

RDD (Resilient Distributed Dataset) is a fundamental data structure of Spark and it is the primary data abstraction in Apache Spark and the Spark Core. RDDs are fault-tolerant, immutable distributed collections of objects, which means once you create an RDD you cannot change it. Each dataset in RDD is divided into logical partitions, which can be computed on different nodes of the cluster.

This Apache Spark RDD Tutorial will help you start understanding and using Apache Spark RDD (Resilient Distributed Dataset) with Scala code examples. All RDD examples provided in this tutorial were also tested in our development environment and are available at GitHub spark scala examples project for quick reference.

In this section of the Apache Spark tutorial, I will introduce the RDD and explains how to create them and use its transformation and action operations. Here is the full article on Spark RDD in case if you wanted to learn more of and get your fundamentals strong.

RDD creation

RDD's are created primarily in two different ways, first parallelizing an existing collection and secondly referencing a dataset in an external storage system (`HDFS`, `HDFS`, `S3` and many more).

sparkContext.parallelize()

sparkContext.parallelize is used to parallelize an existing collection in your driver program. This is a basic method to create RDD.

```
//Create RDD from parallelize
val dataSeq = Seq(("Java", 20000), ("Python", 100000), ("Scala", 3000))
val rdd=spark.sparkContext.parallelize(dataSeq)
```

Copy

sparkContext.textFile()

Using textFile() method we can read a text (.txt) file from many sources like HDFS, S#, Azure, local e.t.c into RDD.

```
//Create RDD from external Data source
val rdd2 = spark.sparkContext.textFile("/path/textFile.txt")
```

Copy

RDD Operations

On Spark RDD, you can perform two kinds of operations.

RDD Transformations

Spark RDD Transformations are lazy operations meaning they don't execute until you call an action on RDD. Since RDD's are immutable, When you run a transformation(for example map()), instead of updating a current RDD, it returns a new RDD.

Some transformations on RDD’s are `flatMap()`, `map()`, `reduceByKey()`, `filter()`, `sortByKey()` and all these return a new RDD instead of updating the current.

RDD Actions

RDD Action operation returns the values from an RDD to a driver node. In other words, any RDD function that returns non RDD[T] is considered as an action. RDD operations trigger the computation and return RDD in a List to the driver program.

Some actions on RDD’s are `count()`, `collect()`, `first()`, `max()`, `reduce()` and more.

RDD Examples

- How to read multiple text files into RDD
- Read CSV file into RDD
- Ways to create an RDD
- Create an empty RDD
- RDD Pair Functions
- Generate DataFrame from RDD

PySpark DataFrame

DataFrame definition is very well explained by Databricks hence I do not want to define it again and confuse you. Below is the definition I took it from Databricks.

DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as structured data files, tables in Hive, external databases, or existing RDDs.

– Databricks

If you are coming from a Python background I would assume you already know what Pandas DataFrame is; PySpark DataFrame is mostly similar to Pandas DataFrame with the exception PySpark DataFrames are distributed in the cluster (meaning the data in DataFrame's are stored in different machines in a cluster) and any operations in PySpark executes in parallel on all machines whereas Panda Dataframe stores and operates on a single machine.

If you have no Python background, I would recommend you learn some basics on Python before you proceeding this Spark tutorial. For now, just know that data in PySpark DataFrame's are stored in different machines in a cluster.

Is PySpark faster than pandas?

Due to parallel execution on all cores on multiple machines, PySpark runs operations faster then pandas. In other words, pandas DataFrames run operations on a single node whereas PySpark runs on multiple machines. To know more read at pandas DataFrame vs PySpark Differences with Examples.

DataFrame creation

The simplest way to create a DataFrame is from a Python list of data. DataFrame can also be created from an RDD and by reading files from several sources.

using createDataFrame()

By using `createDataFrame()` function of the SparkSession you can create a DataFrame.

data = [('James','','Smith','1991-04-01','M',3000),

```
  ('Michael','Rose','','2000-05-19','M',4000),

  ('Robert','','Williams','1978-09-05','M',4000),

  ('Maria','Anne','Jones','1967-12-01','F',4000),

  ('Jen','Mary','Brown','1980-02-17','F',-1)

]



columns = ["firstname","middlename","lastname","dob","gender","salary"]

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

Copy

Since DataFrame's are structure format which contains names and columns, we can get the schema of the DataFrame using `df.printSchema()`

`df.show()` shows the 20 elements from the DataFrame.

```
+---------+----------+--------+----------+------+------+
|firstname|middlename|lastname|dob       |gender|salary|
+---------+----------+--------+----------+------+------+
|James    |          |Smith   |1991-04-01|M     |3000  |
|Michael  |Rose      |        |2000-05-19|M     |4000  |
|Robert   |          |Williams|1978-09-05|M     |4000  |
|Maria    |Anne      |Jones   |1967-12-01|F     |4000  |
|Jen      |Mary      |Brown   |1980-02-17|F     |-1    |
+---------+----------+--------+----------+------+------+
```

Copy

DataFrame operations

Like RDD, DataFrame also has operations like Transformations and Actions.

DataFrame from external data sources

In real-time applications, DataFrames are created from external sources like files from the local system, HDFS, S3 Azure, HBase, MySQL table e.t.c. Below is an example of how to read a CSV file from a local system.

```
df = spark.read.csv("/tmp/resources/zipcodes.csv")

df.printSchema()
```

Copy

Supported file formats

DataFrame has a rich set of API which supports reading and writing several file formats

- csv
- text
- Avro
- Parquet
- tsv
- xml and many more

DataFrame Examples

In this section of the PySpark Tutorial, you will find several Spark examples written in Python that help in your projects.

- Different ways to Create DataFrame in PySpark
- PySpark – Ways to Rename column on DataFrame
- PySpark withColumn() usage with Examples
- PySpark – How to Filter data from DataFrame
- PySpark orderBy() and sort() explained
- PySpark explode array and map columns to rows
- PySpark – explode nested array into rows
- PySpark Read CSV file into DataFrame
- PySpark Groupby Explained with Examples
- PySpark Aggregate Functions with Examples
- PySpark Joins Explained with Examples

PySpark SQL Tutorial

`PySpark SQL` is one of the most used PySpark modules which is used for processing structured columnar data format. Once you have a DataFrame created, you can interact with the data by using SQL syntax.

In other words, Spark SQL brings native RAW SQL queries on Spark meaning you can run traditional ANSI SQL's on Spark Dataframe, in the later section of this PySpark SQL tutorial, you will learn in detail using SQL `select`, `where`, `group by`, `join`, `union` e.t.c

In order to use SQL, first, create a temporary table on DataFrame using `createOrReplaceTempView()` function. Once created, this table can be accessed throughout the SparkSession using `sql()` and it will be dropped along with your SparkContext termination.

Use `sql()` method of the SparkSession object to run the query and this method returns a new DataFrame.

```
df.createOrReplaceTempView("PERSON_DATA")

df2 = spark.sql("SELECT * from PERSON_DATA")

df2.printSchema()

df2.show()
```

Copy

Let's see another pyspark example using `group by`.

```
groupDF = spark.sql("SELECT gender, count(*) from PERSON_DATA group by gender")

groupDF.show()
```

Copy

This yields the below output

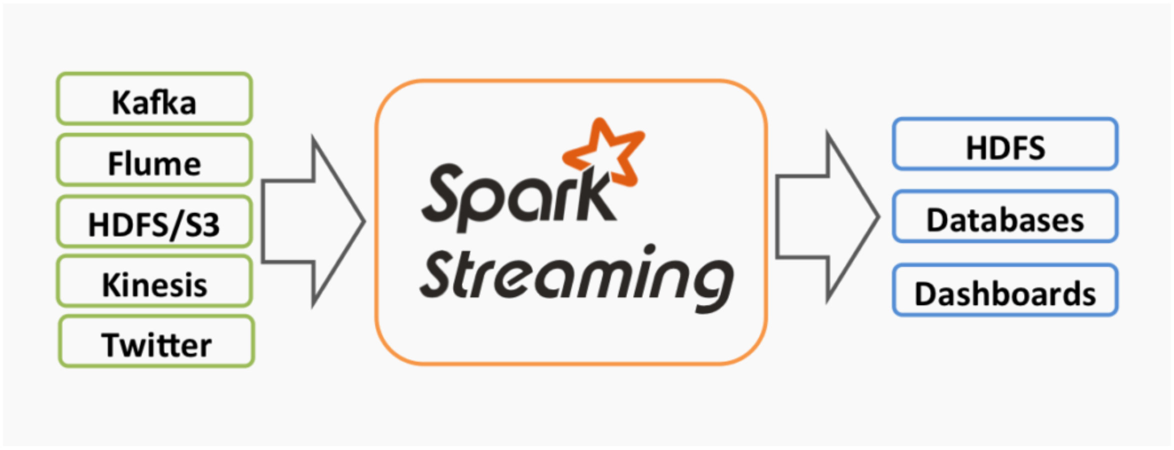
```
+------+--------+

|gender|count(1)|

+------+--------+

|     F|       2|

|     M|       3|

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

Copy

Similarly, you can run any traditional SQL queries on DataFrame's using PySpark SQL.

PySpark Streaming Tutorial

PySpark Streaming is a scalable, high-throughput, fault-tolerant streaming processing system that supports both batch and streaming workloads. It is used to process real-time data from sources like file system folder, TCP socket, S3, Kafka, Flume, Twitter, and Amazon Kinesis to name a few. The processed data can be pushed to databases, Kafka, live dashboards e.t.c



source: https://spark.apache.org/

Streaming from TCP Socket

Use `readStream.format("socket")` from Spark session object to read data from the socket and provide options host and port where you want to stream data from.

```
df = spark.readStream

      .format("socket")

      .option("host","localhost")

      .option("port","9090")

      .load()
```

Copy

Spark reads the data from the socket and represents it in a "value" column of DataFrame. `df.printSchema()` outputs

```
root

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

Copy

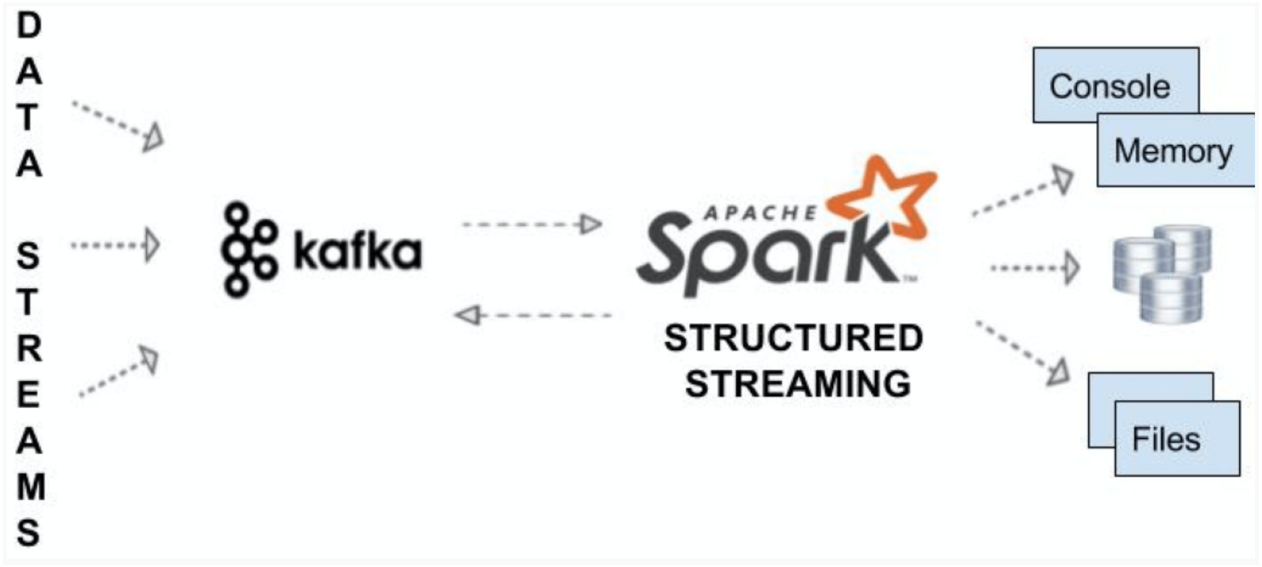
After processing, you can stream the DataFrame to console. In real-time, we ideally stream it to either Kafka, database e.t.c

```
query = count.writeStream

      .format("console")

      .outputMode("complete")

      .start()

      .awaitTermination()
```

Copy

Streaming from Kafka

Using Spark Streaming we can read from Kafka topic and write to Kafka topic in TEXT, CSV, AVRO and JSON formats



```
df = spark.readStream

        .format("kafka")

        .option("kafka.bootstrap.servers", "192.168.1.100:9092")

        .option("subscribe", "json_topic")

        .option("startingOffsets", "earliest") // From starting

        .load()
```

Copy

Below pyspark example, writes message to another topic in Kafka using `writeStream()`

```
df.selectExpr("CAST(id AS STRING) AS key", "to_json(struct(*)) AS value")

   .writeStream

   .format("kafka")

   .outputMode("append")

   .option("kafka.bootstrap.servers", "192.168.1.100:9092")

   .option("topic", "josn_data_topic")

   .start()

   .awaitTermination()
```

PySpark RDD Tutorial | Learn with Examples

This PySpark RDD Tutorial will help you understand what is RDD (Resilient Distributed Dataset)?, It's advantages, how to create, and using it with Github examples. All RDD examples provided in this Tutorial were tested in our development environment and are available at GitHub PySpark examples project for quick reference.

By the end of this PySpark tutorial, you will learn What is PySpark RDD? It's advantages, limitations, creating an RDD, applying transformations, actions, and operating on pair RDD.

- What is PySpark RDD?
- PySpark RDD Benefits
- PySpark RDD Limitations
- Creating RDD
    - Using parallelize()
    - Using textFile()
    - Using wholeTextFiles()
    - create empty RDD

- RDD Parallelize
- Repartition and Coalesce
- RDD Operations
    - RDD Transformations
    - RDD Actions
- RDD Types
- Shuffle Operations
- RDD Persistence
    - RDD cache()
    - RDD persist()
    - RDD unpersist()
- PySpark Shared Variables
    - Broadcast Variables
    - Accumulators
- Advanced API – DataFrame & DataSet

What is RDD (Resilient Distributed Dataset)?

RDD (Resilient Distributed Dataset) is a fundamental building block of PySpark which is fault-tolerant, immutable distributed collections of objects. Immutable meaning once you create an RDD you cannot change it. Each record in RDD is divided into logical partitions, which can be computed on different nodes of the cluster.

In other words, RDDs are a collection of objects similar to list in Python, with the difference being RDD is computed on several processes scattered across multiple physical servers also called nodes in a cluster while a Python collection lives and process in just one process.

Additionally, RDDs provide data abstraction of partitioning and distribution of the data designed to run computations in parallel on several nodes, while doing transformations on RDD we don't have to worry about the parallelism as PySpark by default provides.

This Apache PySpark RDD tutorial describes the basic operations available on RDDs, such as `map()`, `filter()`, and `persist()` and many more. In addition, this tutorial also explains Pair RDD functions that operate on RDDs of key-value pairs such as `groupByKey()` and `join()` etc.

Note: RDD’s can have a name and unique identifier (id)

PySpark RDD Benefits

PySpark is widely adapted in Machine learning and Data science community due to it’s advantages compared with traditional python programming.

In-Memory Processing

PySpark loads the data from disk and process in memory and keeps the data in memory, this is the main difference between PySpark and Mapreduce (I/O intensive). In between the transformations, we can also cache/persists the RDD in memory to reuse the previous computations.

Immutability

PySpark RDD’s are immutable in nature meaning, once RDDs are created you cannot modify. When we apply transformations on RDD, PySpark creates a new RDD and maintains the RDD Lineage.

Fault Tolerance

PySpark operates on fault-tolerant data stores on HDFS, S3 e.t.c hence any RDD operation fails, it automatically reloads the data from other partitions. Also, When PySpark applications running on a cluster, PySpark task failures are automatically recovered for a certain number of times (as per the configuration) and finish the application seamlessly.

Lazy Evolution

PySpark does not evaluate the RDD transformations as they appear/encountered by Driver instead it keeps the all transformations as it encounters(DAG) and evaluates the all transformation when it sees the first RDD action.

Partitioning

When you create RDD from a data, It by default partitions the elements in a RDD. By default it partitions to the number of cores available.

PySpark RDD Limitations

PySpark RDDs are not much suitable for applications that make updates to the state store such as storage systems for a web application. For these applications, it is more efficient to use systems that perform traditional update logging and data checkpointing, such as databases. The goal of RDD is to provide an efficient programming model for batch analytics and leave these asynchronous applications.

Creating RDD

RDD's are created primarily in two different ways,

- parallelizing an existing collection and
- referencing a dataset in an external storage system (`HDFS`, `S3` and many more).

Before we look into examples, first let's initialize SparkSession using the builder pattern method defined in SparkSession class. While initializing, we need to provide the master and application name as shown below. In realtime application, you will pass master from spark-submit instead of hardcoding on Spark application.

```
from pyspark.sql import SparkSession

spark:SparkSession = SparkSession.builder()

    .master("local[1]")

    .appName("SparkByExamples.com")
    .getOrCreate()
```

Copy

`master()` – If you are running it on the cluster you need to use your master name as an argument to master(). usually, it would be either yarn (Yet Another Resource Negotiator) or `mesos` depends on your cluster setup.

- **Use `local[x]` when running in Standalone mode. x should be an integer value and should be greater than 0; this represents how many partitions it should create when using RDD, DataFrame, and Dataset. Ideally, x value should be the number of CPU cores you have.**

`appName()` – Used to set your application name.

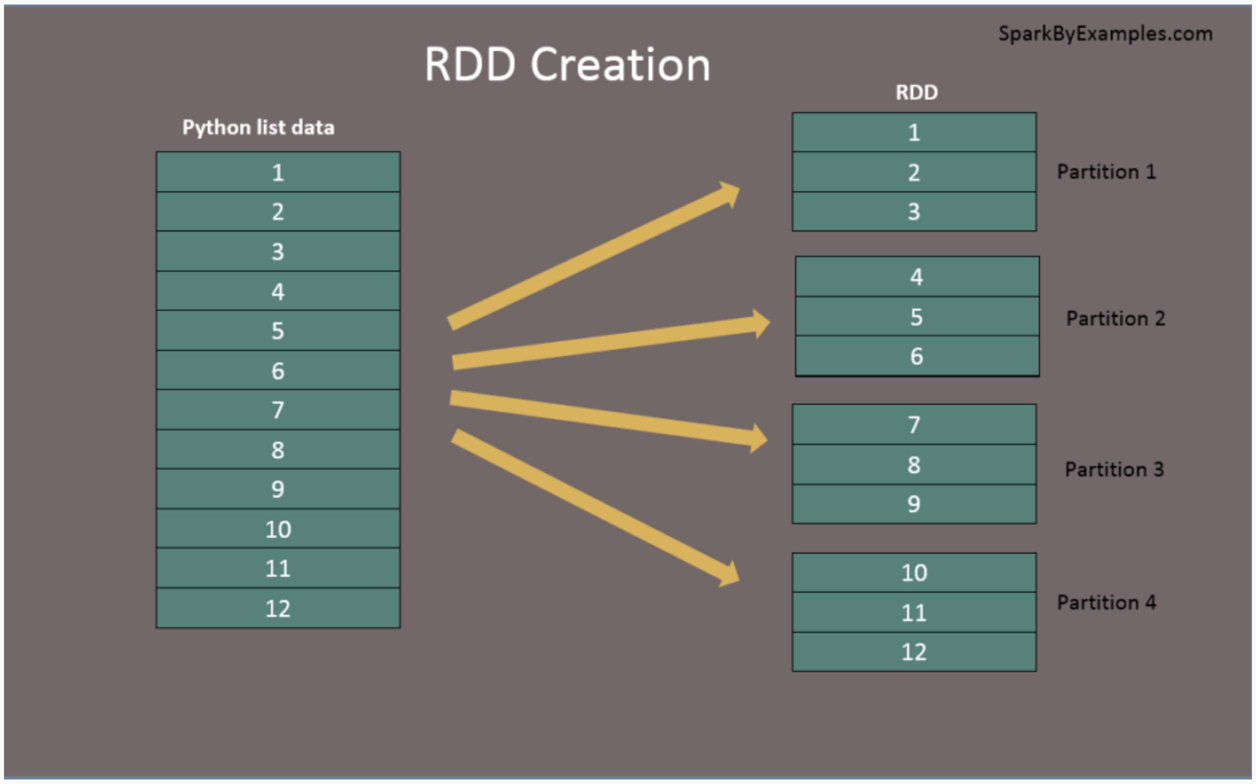
`getOrCreate()` – This returns a SparkSession object if already exists, and creates a new one if not exist.

Note: Creating SparkSession object, internally creates one SparkContext per JVM.

Create RDD using sparkContext.parallelize()

By using `parallelize()` function of SparkContext (sparkContext.parallelize()) you can create an RDD. This function loads the existing collection from your driver program into parallelizing RDD. This is a basic method to create RDD and is used when you already have data in memory that is either loaded from a file or from a

database. and it required all data to be present on the driver program prior to creating RDD.



RDD from list

```
#Create RDD from parallelize

data = [1,2,3,4,5,6,7,8,9,10,11,12]

rdd=spark.sparkContext.parallelize(data)
```

Copy

For production applications, we mostly create RDD by using external storage systems like `HDFS`, `S3`, `HBase` e.t.c. To make it simple for this PySpark RDD tutorial we are using files from the local system or loading it from the python list to create RDD.

Create RDD using sparkContext.textFile()

Using textFile() method we can read a text (.txt) file into RDD.

```
#Create RDD from external Data source

rdd2 = spark.sparkContext.textFile("/path/textFile.txt")
```

Copy

Create RDD using sparkContext.wholeTextFiles()

wholeTextFiles() function returns a PairRDD with the key being the file path and value being file content.

```
#Reads entire file into a RDD as single record.

rdd3 = spark.sparkContext.wholeTextFiles("/path/textFile.txt")
```

Copy

Besides using text files, we can also create RDD from CSV file, JSON, and more formats.

Create empty RDD using sparkContext.emptyRDD

Using `emptyRDD()` method on sparkContext we can create an RDD with no data. This method creates an empty RDD with no partition.

```
# Creates empty RDD with no partition

rdd = spark.sparkContext.emptyRDD

# rddString = spark.sparkContext.emptyRDD[String]
```

Copy

Creating empty RDD with partition

Sometimes we may need to write an empty RDD to files by partition, In this case, you should create an empty RDD with partition.

```
#Create empty RDD with partition

rdd2 = spark.sparkContext.parallelize([],10) #This creates 10 partitions
```

Copy

RDD Parallelize

When we use `parallelize()` or `textFile()` or `wholeTextFiles()` methods of SparkContxt to initiate RDD, it automatically splits the data into partitions based on resource availability. when you run it on a laptop it would create partitions as the same number of cores available on your system.

getNumPartitions() – This a RDD function which returns a number of partitions our dataset split into.

```
print("initial partition count:"+str(rdd.getNumPartitions()))

#Outputs: initial partition count:2
```

Copy

Set parallelize manually – We can also set a number of partitions manually, all, we need is, to pass a number of partitions as the second parameter to these functions for example `sparkContext.parallelize([1,2,3,4,56,7,8,9,12,3], 10)`.

Repartition and Coalesce

Sometimes we may need to repartition the RDD, PySpark provides two ways to repartition; first using `repartition()` method which shuffles data from all nodes also called full shuffle and second coalesce() method which shuffle data from minimum nodes, for examples if you have data in 4 partitions and doing `coalesce(2)` moves data from just 2 nodes.

Both of the functions take the number of partitions to repartition rdd as shown below. Note that repartition() method is a very expensive operation as it shuffles data from all nodes in a cluster.

```
reparRdd = rdd.repartition(4)

print("re-partition count:"+str(reparRdd.getNumPartitions()))

#Outputs: "re-partition count:4
```

Copy

Note: repartition() or coalesce() methods also returns a new RDD.

PySpark RDD Operations

RDD transformations – Transformations are lazy operations, instead of updating an RDD, these operations return another RDD.

RDD actions – operations that trigger computation and return RDD values.

RDD Transformations with example

Transformations on PySpark RDD returns another RDD and transformations are lazy meaning they don't execute until you call an action on RDD. Some transformations on RDD's are `flatMap()`, `map()`, `reduceByKey()`, `filter()`, `sortByKey()` and return new RDD instead of updating the current.

In this PySpark RDD Transformation section of the tutorial, I will explain transformations using the word count example. The below image demonstrates different RDD transformations we going to use.

First, create an RDD by reading a text file. The text file used here is available at the GitHub project.

```
rdd = spark.sparkContext.textFile("/tmp/test.txt")
```

Copy

flatMap – **`flatMap()` transformation flattens the RDD after applying the function and returns a new RDD. On the below example, first, it splits each record by space in an RDD and finally flattens it. Resulting RDD consists of a single word on each record.**

```
rdd2 = rdd.flatMap(lambda x: x.split(" "))
```

Copy

map – `map()` **transformation is used the apply any complex operations like adding a column, updating a column e.t.c, the output of map transformations would always have the same number of records as input.**

In our word count example, we are adding a new column with value 1 for each word, the result of the RDD is `PairRDDFunctions` which contains key-value pairs, word of type String as Key and 1 of type Int as value.

```
rdd3 = rdd2.map(lambda x: (x,1))
```

Copy

reduceByKey – `reduceByKey()` **merges the values for each key with the function specified. In our example, it reduces the word string by applying the sum function on value. The result of our RDD contains unique words and their count.**

```
rdd5 = rdd4.reduceByKey(lambda a,b: a+b)
```

Copy

sortByKey – `sortByKey()` **transformation is used to sort RDD elements on key. In our example, first, we convert RDD[(String,Int]) to RDD[(Int, String]) using map transformation and apply sortByKey which ideally does sort on an integer value. And finally, foreach with println statements returns all words in RDD and their count as key-value pair**

```
rdd6 = rdd5.map(lambda x: (x[1],x[0])).sortByKey()

#Print rdd6 result to console

print(rdd6.collect())
```

Copy

filter – **`filter()` transformation is used to filter the records in an RDD. In our example we are filtering all words starts with "a".**

```
rdd4 = rdd3.filter(lambda x : 'an' in x[1])

print(rdd4.collect())
```

Copy

Please refer to this page for the full list of RDD transformations.

RDD Actions with example

RDD Action operations return the values from an RDD to a driver program. In other words, any RDD function that returns non-RDD is considered as an action.

In this section of the PySpark RDD tutorial, we will continue to use our word count example and performs some actions on it.

count() – Returns the number of records in an RDD

# Action - count

print("Count : "+str(rdd6.count()))

Copy

first() – Returns the first record.

# Action - first

firstRec = rdd6.first()

print("First Record : "+str(firstRec[0]) + ","+ firstRec[1])

Copy

max() – Returns max record.

# Action - max

datMax = rdd6.max()

print("Max Record : "+str(datMax[0]) + ","+ datMax[1])

Copy

reduce() – Reduces the records to single, we can use this to count or sum.

```
# Action - reduce

totalWordCount = rdd6.reduce(lambda a,b: (a[0]+b[0],a[1]))

print("dataReduce Record : "+str(totalWordCount[0]))
```

Copy

take() – Returns the record specified as an argument.

```
# Action - take

data3 = rdd6.take(3)

for f in data3:

    print("data3 Key:"+ str(f[0]) +", Value:"+f[1])
```

Copy

collect() – Returns all data from RDD as an array. Be careful when you use this action when you are working with huge RDD with millions and billions of data as you may run out of memory on the driver.

Action - collect

data = rdd6.collect()

for f in data:

    print("Key:"+ str(f[0]) +", Value:"+f[1])

Copy

saveAsTextFile() – Using saveAsTestFile action, we can write the RDD to a text file.

rdd6.saveAsTextFile("/tmp/wordCount")

Copy

Note: Please refer to this page for a full list of RDD actions.

Types of RDD

PairRDDFunctions or PairRDD – **Pair RDD is a key-value pair This is mostly used RDD type,**

ShuffledRDD –

DoubleRDD –

SequenceFileRDD –

HadoopRDD –

ParallelCollectionRDD –

Shuffle Operations

Shuffling is a mechanism PySpark uses to redistribute the data across different executors and even across machines. PySpark shuffling triggers when we perform certain transformation operations like `gropByKey()`, `reduceByKey()`, `join()` on RDDS

PySpark Shuffle is an expensive operation since it involves the following

- Disk I/O
- Involves data serialization and deserialization
- Network I/O

When creating an RDD, PySpark doesn’t necessarily store the data for all keys in a partition since at the time of creation there is no way we can set the key for data set.

Hence, when we run the `reduceByKey()` operation to aggregate the data on keys, PySpark does the following. needs to first run tasks to collect all the data from all partitions and

For example, when we perform `reduceByKey()` operation, PySpark does the following

- PySpark first runs m*ap* tasks on all partitions which groups all values for a single key.
- The results of the map tasks are kept in memory.
- When results do not fit in memory, PySpark stores the data into a disk.
- PySpark shuffles the mapped data across partitions, some times it also stores the shuffled data into a disk for reuse when it needs to recalculate.
- Run the garbage collection
- Finally runs reduce tasks on each partition based on key.

PySpark RDD triggers shuffle and repartition for several operations like `repartition()` and `coalesce()`, `groupByKey()`, `reduceByKey()`, `cogroup()` and `join()` but not `countByKey()` .

Shuffle partition size & Performance

Based on your dataset size, a number of cores and memory PySpark shuffling can benefit or harm your jobs. When you dealing with less amount of data, you should typically reduce the shuffle partitions otherwise you will end up with many partitioned files with less number of records in each partition. which results in running many tasks with lesser data to process.

On other hand, when you have too much of data and having less number of partitions results in fewer longer running tasks and some times you may also get out of memory error.

Getting the right size of the shuffle partition is always tricky and takes many runs with different values to achieve the optimized number. This is one of the key properties to look for when you have performance issues on PySpark jobs.

PySpark RDD Persistence Tutorial

PySpark *Cache* and P*ersist* are optimization techniques to improve the performance of the RDD jobs that are iterative and interactive. In this PySpark RDD Tutorial section, I will explain how to use persist() and cache() methods on RDD with examples.

Though PySpark provides computation 100 x times faster than traditional Map Reduce jobs, If you have not designed the jobs to reuse the repeating computations you will see degrade in performance when you are dealing with billions or trillions of data. Hence, we need to look at the computations and use optimization techniques as one of the ways to improve performance.

Using `cache()` and `persist()` methods, PySpark provides an optimization mechanism to store the intermediate computation of an RDD so they can be reused in subsequent actions.

When you persist or cache an RDD, each worker node stores it's partitioned data in memory or disk and reuses them in other actions on that RDD. And Spark's persisted data on nodes are fault-tolerant meaning if any partition is lost, it will automatically be recomputed using the original transformations that created it.

Advantages of Persisting RDD

- Cost efficient – PySpark computations are very expensive hence reusing the computations are used to save cost.
- Time efficient – Reusing the repeated computations saves lots of time.
- Execution time – Saves execution time of the job which allows us to perform more jobs on the same cluster.

RDD Cache

PySpark RDD `cache()` method by default saves RDD computation to storage level \``MEMORY_ONLY`\` meaning it will store the data in the JVM heap as unserialized objects.

PySpark `cache()` method in RDD class internally calls `persist()` method which in turn uses `sparkSession.sharedState.cacheManager.cacheQuery` to cache the result set of RDD. Let's look at an example.

```
cachedRdd = rdd.cache()
```

Copy

RDD Persist

PySpark persist() method is used to store the RDD to one of the storage levels `MEMORY_ONLY`,`MEMORY_AND_DISK`, `MEMORY_ONLY_SER`, `MEMORY_AND_DISK_SER`, `DISK_ONLY`, `MEMORY_ONLY_2`,`MEMORY_AND_DISK_2` and more.

PySpark persist has two signature first signature doesn't take any argument which by default saves it to `<strong>MEMORY_ONLY</strong>` storage level and the second signature which takes `StorageLevel` as an argument to store it to different storage levels.

```
import pyspark

dfPersist = rdd.persist(pyspark.StorageLevel.MEMORY_ONLY)

dfPersist.show(false)
```

Copy

RDD Unpersist

PySpark automatically monitors every `persist()` and `cache()` calls you make and it checks usage on each node and drops persisted data if not used or by using least-recently-used (LRU) algorithm. You can also manually remove using `unpersist()` method. unpersist() marks the RDD as non-persistent, and remove all blocks for it from memory and disk.

```
  rddPersist2 = rddPersist.unpersist()
```

Copy

unpersist(Boolean) with boolean as argument blocks until all blocks are deleted.

Persistence Storage Levels

All different storage level PySpark supports are available at `org.apache.spark.storage.StorageLevel` class. Storage Level defines how and where to store the RDD.

`MEMORY_ONLY` – This is the default behavior of the RDD `cache()` method and stores the RDD as deserialized objects to JVM memory. When there is no enough memory

available it will not save to RDD of some partitions and these will be re-computed as and when required. This takes more storage but runs faster as it takes few CPU cycles to read from memory.

`MEMORY_ONLY_SER` – This is the same as `MEMORY_ONLY` but the difference being it stores RDD as serialized objects to JVM memory. It takes lesser memory (space-efficient) then MEMORY_ONLY as it saves objects as serialized and takes an additional few more CPU cycles in order to deserialize.

`MEMORY_ONLY_2` – Same as `MEMORY_ONLY` storage level but replicate each partition to two cluster nodes.

`MEMORY_ONLY_SER_2` – Same as `MEMORY_ONLY_SER` storage level but replicate each partition to two cluster nodes.

`MEMORY_AND_DISK` – In this Storage Level, The RDD will be stored in JVM memory as a deserialized objects. When required storage is greater than available memory, it stores some of the excess partitions in to disk and reads the data from disk when it required. It is slower as there is I/O involved.

`MEMORY_AND_DISK_SER` – This is same as `MEMORY_AND_DISK` storage level difference being it serializes the RDD objects in memory and on disk when space not available.

`MEMORY_AND_DISK_2` – Same as `MEMORY_AND_DISK` storage level but replicate each partition to two cluster nodes.

`MEMORY_AND_DISK_SER_2` – Same as `MEMORY_AND_DISK_SER` storage level but replicate each partition to two cluster nodes.

`DISK_ONLY` – In this storage level, RDD is stored only on disk and the CPU computation time is high as I/O involved.

`DISK_ONLY_2` – Same as `DISK_ONLY` storage level but replicate each partition to two cluster nodes.

PySpark Shared Variables Tutorial

In this section of the PySpark RDD tutorial, let’s learn what are the different types of PySpark Shared variables and how they are used in PySpark transformations.

When PySpark executes transformation using `map()` or `reduce()` operations, It executes the transformations on a remote node by using the variables that are shipped with the tasks and these variables are not sent back to PySpark Driver hence there is no capability to reuse and sharing the variables across tasks. PySpark shared variables solve this problem using the below two techniques. PySpark provides two types of shared variables.

- Broadcast variables (read-only shared variable)
- Accumulator variables (updatable shared variables)

Broadcast read-only Variables

Broadcast variables are read-only shared variables that are cached and available on all nodes in a cluster in-order to access or use by the tasks. Instead of sending this data along with every task, PySpark distributes broadcast variables to the machine using efficient broadcast algorithms to reduce communication costs.

One of the best use-case of PySpark RDD Broadcast is to use with lookup data for example zip code, state, country lookups e.t.c

When you run a PySpark RDD job that has the Broadcast variables defined and used, PySpark does the following.

- PySpark breaks the job into stages that have distributed shuffling and actions are executed with in the stage.
- Later Stages are also broken into tasks
- PySpark broadcasts the common data (reusable) needed by tasks within each stage.
- The broadcasted data is cache in serialized format and deserialized before executing each task.

The PySpark Broadcast is created using the `broadcast(v)` method of the SparkContext class. This method takes the argument v that you want to broadcast.

```
broadcastVar = sc.broadcast([0, 1, 2, 3])

broadcastVar.value
```

Copy

Note that broadcast variables are not sent to executors with sc.broadcast(variable) call instead, they will be sent to executors when they are first used.

Refer to PySpark RDD Broadcast shared variable for more detailed example.

Accumulators

PySpark Accumulators are another type shared variable that are only "added" through an associative and commutative operation and are used to perform counters (Similar to Map-reduce counters) or sum operations.

PySpark by default supports creating an accumulator of any numeric type and provides the capability to add custom accumulator types. Programmers can create following accumulators

- **named accumulators**
- **unnamed accumulators**

When you create a named accumulator, you can see them on PySpark web UI under the "Accumulator" tab. On this tab, you will see two tables; the first table "accumulable" – consists of all named accumulator variables and their values. And on the second table "Tasks" – value for each accumulator modified by a task.

Where as unnamed accumulators are not shows on PySpark web UI, For all practical purposes it is suggestable to use named accumulators.

Accumulator variables are created using `SparkContext.longAccumulator(v)`

```
accum = sc.longAccumulator("SumAccumulator")

sc.parallelize([1, 2, 3]).foreach(lambda x: accum.add(x))
```

Copy

PySpark by default provides accumulator methods for long, double and collection types. All these methods are present in SparkContext class and return `LongAccumulator`, `DoubleAccumulator`, and `CollectionAccumulator` respectively.

- Long Accumulator
- Double Accumulator
- Collection Accumulator

Advanced API – DataFrame & DataSet

Creating RDD from DataFrame and vice-versa

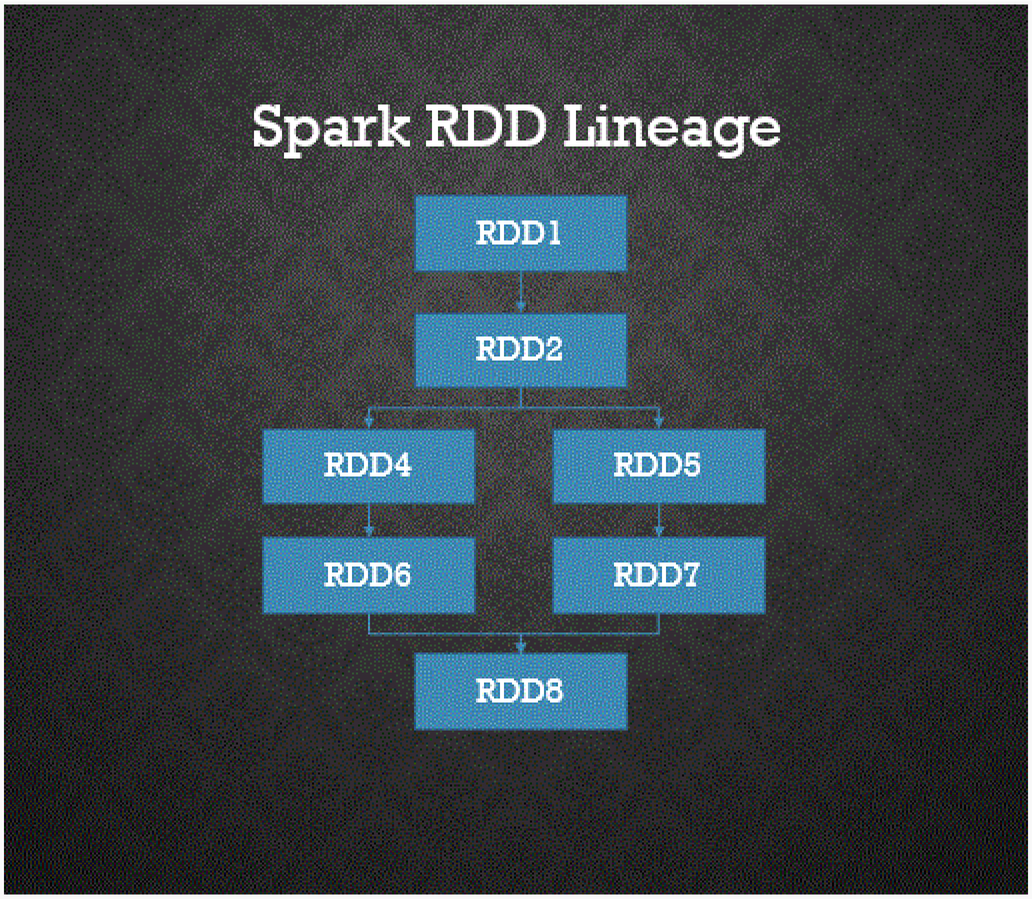
Though we have more advanced API's over RDD, we would often need to convert DataFrame to RDD or RDD to DataFrame. Below are several examples.

```
# Converts RDD to DataFrame

dfFromRDD1 = rdd.toDF()

# Converts RDD to DataFrame with column names

dfFromRDD2 = rdd.toDF("col1","col2")

# using createDataFrame() - Convert DataFrame to RDD

df = spark.createDataFrame(rdd).toDF("col1","col2")

# Convert DataFrame to RDD

rdd = df.rdd
```

Spark RDD Transformations with examples

- Post author:
- NNK
- Post category:
- Apache Spark / Apache Spark RDD

RDD Transformations are Spark operations when executed on RDD, it results in a single or multiple new RDD’s. Since RDD are immutable in nature, transformations always create new RDD without updating an existing one hence, this creates an RDD lineage.



RDD Lineage is also known as the RDD operator graph or RDD dependency graph.

In this tutorial, you will learn lazy transformations, types of transformations, a complete list of transformation functions using wordcount example in scala.

- What is a lazy transformation
- Transformation types
    - Narrow transformation
    - Wider transformation
- Transformation functions
- Transformation functions with word count examples

RDD Transformations are Lazy

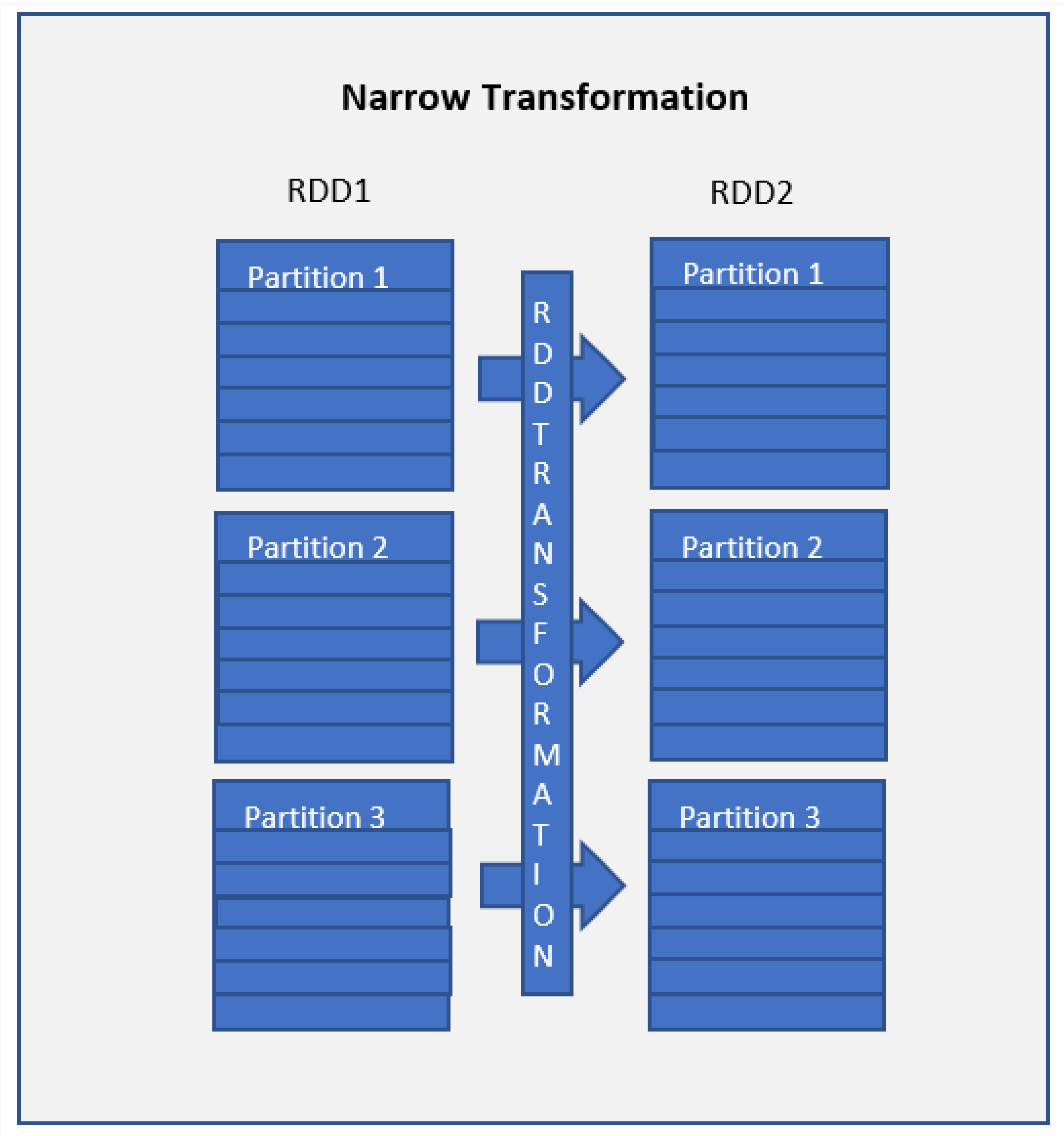
RDD Transformations are lazy operations meaning none of the transformations get executed until you call an action on Spark RDD. Since RDD's are immutable, any transformations on it result in a new RDD leaving the current one unchanged.

RDD Transformation Types

There are two types are transformations.

Narrow Transformation

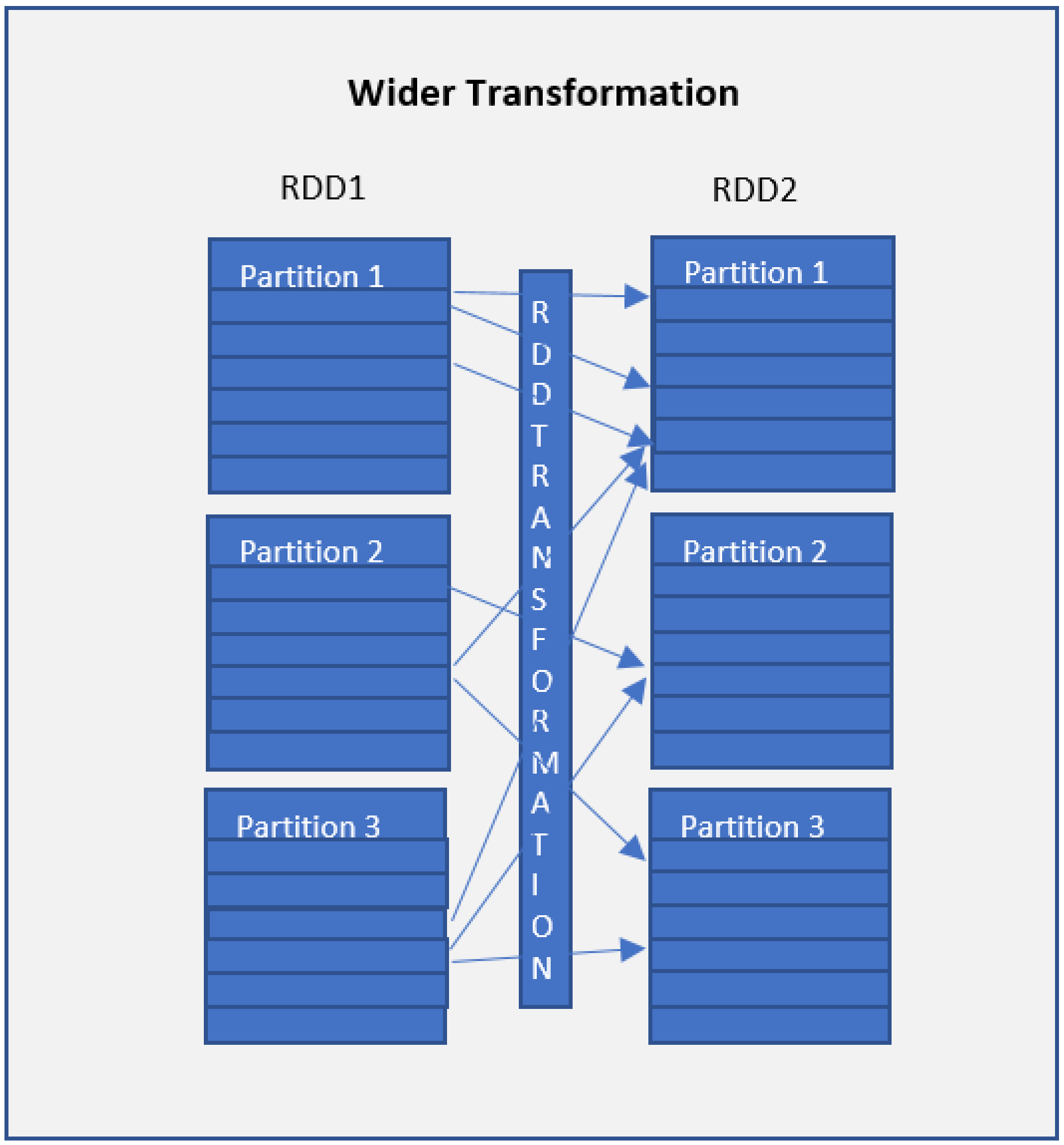
Narrow transformations are the result of map() and filter() functions and these compute data that live on a single partition meaning there will not be any data movement between partitions to execute narrow transformations.



Functions such as `map()`, `mapPartition()`, `flatMap()`, `filter()`, `union()` are some examples of narrow transformation

Wider Transformation

Wider transformations are the result of *groupByKey()* and *reduceByKey()* functions and these compute data that live on many partitions meaning there will be data movements between partitions to execute wider transformations. Since these shuffles the data, they also called shuffle transformations.



Functions such as `groupByKey()`, `aggregateByKey()`, `aggregate()`, `join()`, `repartition()` are some examples of a wider transformations.

Note: When compared to Narrow transformations, wider transformations are expensive operations due to shuffling.

Spark RDD Transformation functions

Search:

TRANSFORMATION METHODS	**METHOD USAGE AND DESCRIPTION**
cache()	Caches the RDD
filter()	Returns a new RDD after applying filter function on source dataset.
flatMap()	Returns flattern map meaning if you have a dataset with array, it converts each elements in a array as a row. In other words it return 0 or more items in output for each element in dataset.
map()	Applies transformation function on dataset and returns same number of elements in distributed dataset.
mapPartitions()	Similar to map, but executs transformation function on each partition, This gives better performance than map function
mapPartitionsWithIndex()	Similar to map Partitions, but also provides func with an integer value representing the index of the partition.
randomSplit()	Splits the RDD by the weights specified in the argument. For example rdd.randomSplit(0.7,0.3)

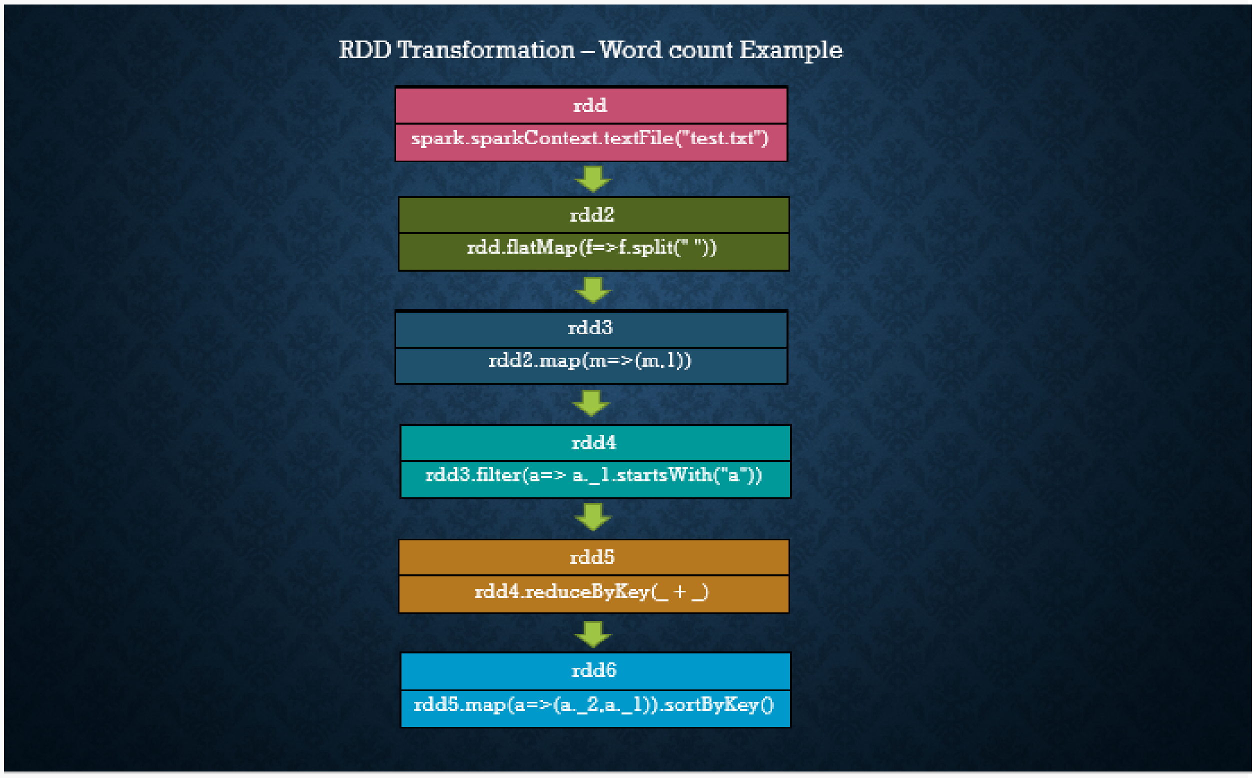
union()	Comines elements from source dataset and the argument and returns combined dataset. This is similar to union function in Math set operations.
sample()	Returns the sample dataset.
intersection()	Returns the dataset which contains elements in both source dataset and an argument
distinct()	Returns the dataset by eliminating all duplicated elements.
repartition()	Return a dataset with number of partition specified in the argument. This operation reshuffles the RDD randamly, It could either return lesser or more partioned RDD based on the input supplied.
coalesce()	Similar to repartition by operates better when we want to the decrease the partitions. Betterment acheives by reshuffling the data from fewer nodes compared with all nodes by repartition.

Spark RDD Transformations with Examples

In this section, I will explain a few RDD Transformations with word count example in scala, before we start first, let's create an RDD by reading a text file. The text file used here is available at the GitHub and, the scala example is available at GitHub project for reference.

```
val spark:SparkSession = SparkSession.builder()
    .master("local[3]")
    .appName("SparkByExamples.com")
    .getOrCreate()

val sc = spark.sparkContext

val rdd:RDD[String] = sc.textFile("src/main/scala/test.txt")
```

Copy


RDD Transformation – Word count Example
rdd
spark.sparkContext.textFile("test.txt")
rdd2
rdd.flatMap(f=>f.split(" "))
rdd3
rdd2.map(m=>(m,1))
rdd4
rdd3.filter(a=> a._1.startsWith("a"))
rdd5
rdd4.reduceByKey(_ + _)
rdd6
rdd5.map(a=>(a._2,a._1)).sortByKey()

flatMap() Transformation

`flatMap()` transformation flattens the RDD after applying the function and returns a new RDD. On the below example, first, it splits each record by space in an RDD and finally flattens it. Resulting RDD consists of a single word on each record.

```
val rdd2 = rdd.flatMap(f=>f.split(" "))
```

Copy

map() Transformation

`map()` transformation is used the apply any complex operations like adding a column, updating a column e.t.c, the output of map transformations would always have the same number of records as input.

In our word count example, we are adding a new column with value 1 for each word, the result of the RDD is PairRDDFunctions which contains key-value pairs, word of type String as Key and 1 of type Int as value. For your understanding, I’ve defined rdd3 variable with type.

```
val rdd3:RDD[(String,Int)]= rdd2.map(m=>(m,1))
```

Copy

filter() Transformation

`filter()` transformation is used to filter the records in an RDD. In our example we are filtering all words starts with “a”.

```
val rdd4 = rdd3.filter(a=> a._1.startsWith("a"))
```

Copy

reduceByKey() Transformation

`reduceByKey()` merges the values for each key with the function specified. In our example, it reduces the word string by applying the sum function on value. The result of our RDD contains unique words and their count.

```
val rdd5 = rdd3.reduceByKey(_ + _)
```

Copy

sortByKey() Transformation

`sortByKey()` transformation is used to sort RDD elements on key. In our example, first, we convert RDD[(String,Int]) to RDD[(Int,String]) using map transformation and apply sortByKey which ideally does sort on an integer value. And finally, foreach with println statement prints all words in RDD and their count as key-value pair to console.

```
val rdd6 = rdd5.map(a=>(a._2,a._1)).sortByKey()

//Print rdd6 result to console
rdd6.foreach(println)
```

Copy

Spark RDD Transformations complete example

```
package com.sparkbyexamples.spark.rdd

import org.apache.spark.rdd.RDD
import org.apache.spark.sql.SparkSession
```

```
object WordCountExample {

  def main(args:Array[String]): Unit = {

    val spark:SparkSession = SparkSession.builder()
      .master("local[3]")
      .appName("SparkByExamples.com")
      .getOrCreate()

    val sc = spark.sparkContext

    val rdd:RDD[String] = sc.textFile("src/main/resources/test.txt")
    println("initial partition count:"+rdd.getNumPartitions)

    val reparRdd = rdd.repartition(4)
    println("re-partition count:"+reparRdd.getNumPartitions)

    //rdd.coalesce(3)

    rdd.collect().foreach(println)

    // rdd flatMap transformation
    val rdd2 = rdd.flatMap(f=>f.split(" "))
    rdd2.foreach(f=>println(f))

    //Create a Tuple by adding 1 to each word
    val rdd3:RDD[(String,Int)]= rdd2.map(m=>(m,1))
    rdd3.foreach(println)

    //Filter transformation
    val rdd4 = rdd3.filter(a=> a._1.startsWith("a"))
    rdd4.foreach(println)

    //ReduceBy transformation
    val rdd5 = rdd3.reduceByKey(_ + _)
    rdd5.foreach(println)

    //Swap word,count and sortByKey transformation
    val rdd6 = rdd5.map(a=>(a._2,a._1)).sortByKey()
    println("Final Result")

    //Action - foreach
    rdd6.foreach(println)
    //Action - count
    println("Count : "+rdd6.count())

    //Action - first
    val firstRec = rdd6.first()
    println("First Record : "+firstRec._1 + ","+ firstRec._2)

    //Action - max
    val datMax = rdd6.max()
    println("Max Record : "+datMax._1 + ","+ datMax._2)

    //Action - reduce
    val totalWordCount = rdd6.reduce((a,b) => (a._1+b._1,a._2))
    println("dataReduce Record : "+totalWordCount._1)
    //Action - take
    val data3 = rdd6.take(3)
    data3.foreach(f=>{
      println("data3 Key:"+ f._1 +", Value:"+f._2)
    })

    //Action - collect
    val data = rdd6.collect()
    data.foreach(f=>{
      println("Key:"+ f._1 +", Value:"+f._2)
    })

    //Action - saveAsTextFile
    rdd5.saveAsTextFile("c:/tmp/wordCount")

  }
}
```

Spark RDD Actions with examples

- Post author:
- Admin

- Post category:
- Apache Spark / Apache Spark RDD

Table of Contents

RDD actions are operations that return the raw values, In other words, any RDD function that returns other than RDD[T] is considered as an action in spark programming. In this tutorial, we will learn RDD actions with Scala examples.

As mentioned in RDD Transformations, all transformations are lazy meaning they do not get executed right away and action functions trigger to execute the transformations.

Complete code I've used in this article is available at GitHub project for quick reference.

Spark RDD Actions

Select a link from the table below to jump to an example.

RDD ACTION METHODS	**METHOD DEFINITION**
aggregate[U](zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): U	Aggregate the elements of each partition, and then the results for all the partitions.
collect():Array[T]	Return the complete dataset as an Array.

count():Long	Return the count of elements in the dataset.
countApprox(timeout: Long, confidence: Double = 0.95): PartialResult[BoundedDouble]	Return approximate count of elements in the dataset, this method returns incomplete when execution time meets timeout.
countApproxDistinct(relativeSD: Double = 0.05): Long	Return an approximate number of distinct elements in the dataset.
countByValue(): Map[T, Long]	Return Map[T,Long] key representing each unique value in dataset and value represent count each value present.
countByValueApprox(timeout: Long, confidence: Double = 0.95)(implicit ord: Ordering[T] = null): PartialResult[Map[T, BoundedDouble]]	Same as countByValue() but returns approximate result.
first():T	Return the first element in the dataset.
fold(zeroValue: T)(op: (T, T) ⇒ T): T	Aggregate the elements of each partition, and then the results for all the partitions.
foreach(f: (T) ⇒ Unit): Unit	Iterates all elements in the dataset by applying function f to all elements.

foreachPartition(f: (Iterator[T]) ⇒ Unit): Unit	Similar to foreach, but applies function f for each partition.
min()(implicit ord: Ordering[T]): T	Return the minimum value from the dataset.
max()(implicit ord: Ordering[T]): T	Return the maximum value from the dataset.
reduce(f: (T, T) ⇒ T): T	Reduces the elements of the dataset using the specified binary operator.
saveAsObjectFile(path: String): Unit	Saves RDD as a serialized object's to the storage system.
saveAsTextFile(path: String, codec: Class[_ <: CompressionCodec]): Unit	Saves RDD as a compressed text file.
saveAsTextFile(path: String): Unit	Saves RDD as a text file.
take(num: Int): Array[T]	Return the first num elements of the dataset.

takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]	Return the first num (smallest) elements from the dataset and this is the opposite of the take() action. Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
takeSample(withReplacement: Boolean, num: Int, seed: Long = Utils.random.nextLong): Array[T]	Return the subset of the dataset in an Array. Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
toLocalIterator(): Iterator[T]	Return the complete dataset as an Iterator. Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
top(num: Int)(implicit ord: Ordering[T]): Array[T]	Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.
treeAggregate	Aggregates the elements of this RDD in a multi-level tree pattern.

treeReduce	Reduces the elements of this RDD in a multi-level tree pattern.

RDD Actions Example

Before we start explaining RDD actions with examples, first, let’s create an RDD.

```
  val spark = SparkSession.builder()
    .appName("SparkByExample")
    .master("local")
    .getOrCreate()

  spark.sparkContext.setLogLevel("ERROR")
  val inputRDD = spark.sparkContext.parallelize(List(("Z", 1),("A", 20),("B", 30),("C", 40),("B",
30),("B", 60)))

  val listRdd = spark.sparkContext.parallelize(List(1,2,3,4,5,3,2))
```

Copy

Note that we have created two RDD’s in the above code snippet and we use these two as and when necessary to demonstrate the RDD actions.

aggregate – action

`aggregate()` the elements of each partition, and then the results for all the partitions.

```
  //aggregate
  def param0= (accu:Int, v:Int) => accu + v
```

```
def param1= (accu1:Int,accu2:Int) => accu1 + accu2
println("aggregate : "+listRdd.aggregate(0)(param0,param1))
//Output: aggregate : 20

//aggregate
def param3= (accu:Int, v:(String,Int)) => accu + v._2
def param4= (accu1:Int,accu2:Int) => accu1 + accu2
println("aggregate : "+inputRDD.aggregate(0)(param3,param4))
//Output: aggregate : 20
```

Copy

treeAggregate – action

`treeAggregate()` – Aggregates the elements of this RDD in a multi-level tree pattern. The output of this function will be similar to the aggregate function.

```
//treeAggregate. This is similar to aggregate
def param8= (accu:Int, v:Int) => accu + v
def param9= (accu1:Int,accu2:Int) => accu1 + accu2
println("treeAggregate : "+listRdd.treeAggregate(0)(param8,param9))
//Output: treeAggregate : 20
```

Copy

fold – action

`fold()` – Aggregate the elements of each partition, and then the results for all the partitions.

```
//fold
println("fold :  "+listRdd.fold(0){ (acc,v) =>
  val sum = acc+v
  sum
})
//Output: fold :  20

println("fold :  "+inputRDD.fold(("Total",0)){(acc:(String,Int),v:(String,Int))=>
  val sum = acc._2 + v._2
  ("Total",sum)
})
//Output: fold :  (Total,181)
```

Copy

reduce

`reduce()` – Reduces the elements of the dataset using the specified binary operator.

```
//reduce
println("reduce : "+listRdd.reduce(_ + _))
//Output: reduce : 20
println("reduce alternate : "+listRdd.reduce((x, y) => x + y))
//Output: reduce alternate : 20
println("reduce : "+inputRDD.reduce((x, y) => ("Total",x._2 + y._2)))
//Output: reduce : (Total,181)
```

Copy

treeReduce

`treeReduce()` – Reduces the elements of this RDD in a multi-level tree pattern.

```
//treeReduce. This is similar to reduce
println("treeReduce : "+listRdd.treeReduce(_ + _))
//Output: treeReduce : 20
```

Copy

collect

`collect()` -Return the complete dataset as an Array.

```
//Collect
val data:Array[Int] = listRdd.collect()
data.foreach(println)
```

Copy

count, countApprox, countApproxDistinct

`count()` – Return the count of elements in the dataset.

`countApprox()` – Return approximate count of elements in the dataset, this method returns incomplete when execution time meets timeout.

`countApproxDistinct()` – Return an approximate number of distinct elements in the dataset.

```
//count, countApprox, countApproxDistinct
```

```
println("Count : "+listRdd.count)
//Output: Count : 20
println("countApprox : "+listRdd.countApprox(1200))
//Output: countApprox : (final: [7.000, 7.000])
println("countApproxDistinct : "+listRdd.countApproxDistinct())
//Output: countApproxDistinct : 5
println("countApproxDistinct : "+inputRDD.countApproxDistinct())
//Output: countApproxDistinct : 5
```

Copy

countByValue, countByValueApprox

`countByValue()` – Return Map[T,Long] key representing each unique value in dataset and value represents count each value present.

`countByValueApprox()` – Same as countByValue() but returns approximate result.

```
//countByValue, countByValueApprox
println("countByValue :  "+listRdd.countByValue())
//Output: countByValue :  Map(5 -> 1, 1 -> 1, 2 -> 2, 3 -> 2, 4 -> 1)
//println(listRdd.countByValueApprox())
```

Copy

first

`first()` – Return the first element in the dataset.

```
//first
println("first :  "+listRdd.first())
//Output: first :  1
println("first :  "+inputRDD.first())
//Output: first :  (Z,1)
```

Copy

top

`top()` – Return top n elements from the dataset.

Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.

```
//top
println("top : "+listRdd.top(2).mkString(","))
//Output: take : 5,4
println("top : "+inputRDD.top(2).mkString(","))
//Output: take : (Z,1),(C,40)
```

Copy

min

`min()` – Return the minimum value from the dataset.

```
//min
```

```
println("min :  "+listRdd.min())
//Output: min :  1
println("min :  "+inputRDD.min())
//Output: min :  (A,20)
```

Copy

max

`max()` – Return the maximum value from the dataset.

```
//max
println("max :  "+listRdd.max())
//Output: max :  5
println("max :  "+inputRDD.max())
//Output: max :  (Z,1)
```

Copy

take, takeOrdered, takeSample

`take()` – Return the first num elements of the dataset.

`takeOrdered()` – Return the first num (smallest) elements from the dataset and this is the opposite of the take() action.

Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.

`takeSample()` – Return the subset of the dataset in an Array.

Note: Use this method only when the resulting array is small, as all the data is loaded into the driver's memory.

```
  //take, takeOrdered, takeSample
  println("take : "+listRdd.take(2).mkString(","))
  //Output: take : 1,2
  println("takeOrdered : "+ listRdd.takeOrdered(2).mkString(","))
  //Output: takeOrdered : 1,2
  //println("take : "+listRdd.takeSample())
```

Copy

Actions – Complete example

```
package com.sparkbyexamples.spark.rdd

import com.sparkbyexamples.spark.rdd.OperationOnPairRDDComplex.kv
import org.apache.spark.sql.SparkSession

import scala.collection.mutable

object RDDActions extends App {

  val spark = SparkSession.builder()
    .appName("SparkByExample")
    .master("local")
    .getOrCreate()

  spark.sparkContext.setLogLevel("ERROR")
  val inputRDD = spark.sparkContext.parallelize(List(("Z", 1),("A", 20),("B", 30),("C", 40),("B", 
30),("B", 60)))
```

```
val listRdd = spark.sparkContext.parallelize(List(1,2,3,4,5,3,2))

//Collect
val data:Array[Int] = listRdd.collect()
data.foreach(println)

//aggregate
def param0= (accu:Int, v:Int) => accu + v
def param1= (accu1:Int,accu2:Int) => accu1 + accu2
println("aggregate : "+listRdd.aggregate(0)(param0,param1))
//Output: aggregate : 20

//aggregate
def param3= (accu:Int, v:(String,Int)) => accu + v._2
def param4= (accu1:Int,accu2:Int) => accu1 + accu2
println("aggregate : "+inputRDD.aggregate(0)(param3,param4))
//Output: aggregate : 20

//treeAggregate. This is similar to aggregate
def param8= (accu:Int, v:Int) => accu + v
def param9= (accu1:Int,accu2:Int) => accu1 + accu2
println("treeAggregate : "+listRdd.treeAggregate(0)(param8,param9))
//Output: treeAggregate : 20

//fold
println("fold :  "+listRdd.fold(0){ (acc,v) =>
  val sum = acc+v
  sum
})
//Output: fold :  20

println("fold :  "+inputRDD.fold(("Total",0)){(acc:(String,Int),v:(String,Int))=>
  val sum = acc._2 + v._2
  ("Total",sum)
})
//Output: fold :  (Total,181)

//reduce
println("reduce : "+listRdd.reduce(_ + _))
//Output: reduce : 20
println("reduce alternate : "+listRdd.reduce((x, y) => x + y))
//Output: reduce alternate : 20
println("reduce : "+inputRDD.reduce((x, y) => ("Total",x._2 + y._2)))
//Output: reduce : (Total,181)
```

```
//treeReduce. This is similar to reduce
println("treeReduce : "+listRdd.treeReduce(_ + _))
//Output: treeReduce : 20

//count, countApprox, countApproxDistinct
println("Count : "+listRdd.count)
//Output: Count : 20
println("countApprox : "+listRdd.countApprox(1200))
//Output: countApprox : (final: [7.000, 7.000])
println("countApproxDistinct : "+listRdd.countApproxDistinct())
//Output: countApproxDistinct : 5
println("countApproxDistinct : "+inputRDD.countApproxDistinct())
//Output: countApproxDistinct : 5

//countByValue, countByValueApprox
println("countByValue :  "+listRdd.countByValue())
//Output: countByValue :  Map(5 -> 1, 1 -> 1, 2 -> 2, 3 -> 2, 4 -> 1)
//println(listRdd.countByValueApprox())

//first
println("first :  "+listRdd.first())
//Output: first :  1
println("first :  "+inputRDD.first())
//Output: first :  (Z,1)

//top
println("top : "+listRdd.top(2).mkString(","))
//Output: take : 5,4
println("top : "+inputRDD.top(2).mkString(","))
//Output: take : (Z,1),(C,40)

//min
println("min :  "+listRdd.min())
//Output: min :  1
println("min :  "+inputRDD.min())
//Output: min :  (A,20)

//max
println("max :  "+listRdd.max())
//Output: max :  5
println("max :  "+inputRDD.max())
//Output: max :  (Z,1)
```

```
 //take, takeOrdered, takeSample
 println("take : "+listRdd.take(2).mkString(","))
 //Output: take : 1,2
 println("takeOrdered : "+ listRdd.takeOrdered(2).mkString(","))
 //Output: takeOrdered : 1,2
 //println("take : "+listRdd.takeSample())

 //toLocalIterator
 //listRdd.toLocalIterator.foreach(println)
 //Output:

}
```

PySpark – Create DataFrame with Examples

- Post author:
- NNK
- Post category:
- PySpark

Table of Contents

You can manually create a PySpark DataFrame using `toDF()` and `createDataFrame()` methods, both these function takes different signatures in order to create DataFrame from existing RDD, list, and DataFrame.

You can also create PySpark DataFrame from data sources like TXT, CSV, JSON, ORV, Avro, Parquet, XML formats by reading from HDFS, S3, DBFS, Azure Blob file systems e.t.c.

Related:

- [Fetch More Than 20 Rows & Column Full Value in DataFrame]

- Get Current Number of Partitions of Spark DataFrame
- How to check if Column Present in Spark DataFrame

Finally, PySpark DataFrame also can be created by reading data from RDBMS Databases and NoSQL databases.

In this article, you will learn to create DataFrame by some of these methods with PySpark examples.

Table of Contents

- Create DataFrame from RDD
    - toDF()
    - createDataFrame()
- Create DataFrame from the list of data
- Create DataFrame from Data sources
    - Creating from CSV file
    - Creating from TXT file
    - Creating from JSON file
- Other sources (Avro, Parquet, ORC e.t.c)

SPARKSESSION	RDD	DATAFRAME
createDataFrame(rdd)	toDF()	toDF(*cols)

createDataFrame(dataList)	toDF(*cols)	
createDataFrame(rowData,columns)		
createDataFrame(dataList,schema)		

PySpark Create DataFrame matrix

In order to create a DataFrame from a list we need the data hence, first, let's create the data and the columns that are needed.

```
columns = ["language","users_count"]
data = [("Java", "20000"), ("Python", "100000"), ("Scala", "3000")]
```

Copy

1. Create DataFrame from RDD

One easy way to manually create PySpark DataFrame is from an existing RDD. first, let's create a Spark RDD from a collection List by calling parallelize() function from SparkContext . We would need this rdd object for all our examples below.

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

```
rdd = spark.sparkContext.parallelize(data)
```

Copy

1.1 Using toDF() function

PySpark RDD's toDF() method is used to create a DataFrame from the existing RDD. Since RDD doesn't have columns, the DataFrame is created with default column names "_1" and "_2" as we have two columns.

```
dfFromRDD1 = rdd.toDF()
dfFromRDD1.printSchema()
```

Copy

PySpark printschema() yields the schema of the DataFrame to console.

```
root
 |-- _1: string (nullable = true)
 |-- _2: string (nullable = true)
```

Copy

If you wanted to provide column names to the DataFrame use `toDF()` method with column names as arguments as shown below.

```
columns = ["language","users_count"]
dfFromRDD1 = rdd.toDF(columns)
dfFromRDD1.printSchema()
```

Copy

This yields the schema of the DataFrame with column names. use the show() method on PySpark DataFrame to show the DataFrame

```
root
 |-- language: string (nullable = true)
 |-- users: string (nullable = true)
```

Copy

By default, the datatype of these columns infers to the type of data. We can change this behavior by supplying schema, where we can specify a column name, data type, and nullable for each field/column.

1.2 Using createDataFrame() from SparkSession

Using createDataFrame() from SparkSession is another way to create manually and it takes rdd object as an argument. and chain with toDF() to specify name to the columns.

```
dfFromRDD2 = spark.createDataFrame(rdd).toDF(*columns)
```

Copy

2. Create DataFrame from List Collection

In this section, we will see how to create PySpark DataFrame from a list. These examples would be similar to what we have seen in the above section with RDD, but we use the list data object instead of “rdd” object to create DataFrame.

2.1 Using createDataFrame() from SparkSession

Calling `createDataFrame()` from `SparkSession` is another way to create PySpark DataFrame manually, it takes a list object as an argument. and chain with `toDF()` to specify names to the columns.

```
dfFromData2 = spark.createDataFrame(data).toDF(*columns)
```

Copy

2.2 Using createDataFrame() with the Row type

`createDataFrame()` has another signature in PySpark which takes the collection of Row type and schema for column names as arguments. To use this first we need to convert our “data” object from the list to list of Row.

```
rowData = map(lambda x: Row(*x), data)
dfFromData3 = spark.createDataFrame(rowData,columns)
```

Copy

2.3 Create DataFrame with schema

If you wanted to specify the column names along with their data types, you should create the StructType schema first and then assign this while creating a DataFrame.

```
from pyspark.sql.types import StructType,StructField, StringType, IntegerType
data2 = [("James","","Smith","36636","M",3000),
    ("Michael","Rose","","40288","M",4000),
    ("Robert","","Williams","42114","M",4000),
    ("Maria","Anne","Jones","39192","F",4000),
    ("Jen","Mary","Brown","","F",-1)
  ]

schema = StructType([ \
    StructField("firstname",StringType(),True), \
    StructField("middlename",StringType(),True), \
    StructField("lastname",StringType(),True), \
    StructField("id", StringType(), True), \
    StructField("gender", StringType(), True), \
    StructField("salary", IntegerType(), True) \
  ])

df = spark.createDataFrame(data=data2,schema=schema)
df.printSchema()
df.show(truncate=False)
```

Copy

This yields below output.

```
root
 |-- firstname: string (nullable = true)
 |-- middlename: string (nullable = true)
 |-- lastname: string (nullable = true)
 |-- id: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- salary: integer (nullable = true)

+---------+----------+--------+-----+------+------+
|firstname|middlename|lastname|id   |gender|salary|
+---------+----------+--------+-----+------+------+
|James    |          |Smith   |36636|M     |3000  |
|Michael  |Rose      |        |40288|M     |4000  |
|Robert   |          |Williams|42114|M     |4000  |
|Maria    |Anne      |Jones   |39192|F     |4000  |
|Jen      |Mary      |Brown   |     |F     |-1    |
+---------+----------+--------+-----+------+------+
```

3. Create DataFrame from Data sources

In real-time mostly you create DataFrame from data source files like CSV, Text, JSON, XML e.t.c.

PySpark by default supports many data formats out of the box without importing any libraries and to create DataFrame you need to use the appropriate method available in `DataFrameReader` class.

3.1 Creating DataFrame from CSV

Use `csv()` method of the `DataFrameReader` object to create a DataFrame from CSV file. you can also provide options like what delimiter to use, whether you have quoted data, date formats, infer schema, and many more. Please refer PySpark Read CSV into DataFrame

```
df2 = spark.read.csv("/src/resources/file.csv")
```

Copy

3.2. Creating from text (TXT) file

Similarly you can also create a DataFrame by reading a from Text file, use `text()` method of the DataFrameReader to do so.

```
df2 = spark.read.text("/src/resources/file.txt")
```

Copy

3.3. Creating from JSON file

PySpark is also used to process semi-structured data files like JSON format. you can use `json()` method of the DataFrameReader to read JSON file into DataFrame. Below is a simple example.

```
df2 = spark.read.json("/src/resources/file.json")
```

Copy

Similarly, we can create DataFrame in PySpark from most of the relational databases which I've not covered here and I will leave this to you to explore.

4. Other sources (Avro, Parquet, ORC, Kafka)

We can also create DataFrame by reading Avro, Parquet, ORC, Binary files and accessing Hive and HBase table, and also reading data from Kafka which I've explained in the below articles, I would recommend reading these when you have time.

- PySpark Read Parquet file into DataFrame
- DataFrame from Avro source
- DataFrame by Streaming data from Kafka

The complete code can be downloaded from GitHub

Happy Learning !!

PySpark withColumn() Usage with Examples

- Post author:
- NNK
- Post category:
- PySpark

Table of Contents

PySpark `withColumn()` is a transformation function of DataFrame which is used to change the value, convert the datatype of an existing column, create a new column, and many more. In this post, I will walk you through commonly used PySpark DataFrame column operations using withColumn() examples.

- PySpark withColumn – To change column DataType
- Transform/change value of an existing column
- Derive new column from an existing column
- Add a column with the literal value
- Rename column name
- Drop DataFrame column

First, let's create a DataFrame to work with.

```
data = [('James','','Smith','1991-04-01','M',3000),
  ('Michael','Rose','','2000-05-19','M',4000),
  ('Robert','','Williams','1978-09-05','M',4000),
  ('Maria','Anne','Jones','1967-12-01','F',4000),
  ('Jen','Mary','Brown','1980-02-17','F',-1)
]

columns = ["firstname","middlename","lastname","dob","gender","salary"]
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
df = spark.createDataFrame(data=data, schema = columns)
```

Copy

1. Change DataType using PySpark withColumn()

By using PySpark `withColumn()` on a DataFrame, we can cast or change the data type of a column. In order to change data type, you would also need to use `cast()` function along with withColumn(). The below statement changes the datatype from `String` to `Integer` for the `salary` column.

```
df.withColumn("salary",col("salary").cast("Integer")).show()
```

Copy

2. Update The Value of an Existing Column

PySpark `withColumn()` function of DataFrame can also be used to change the value of an existing column. In order to change the value, pass an existing column name as a first argument and a value to be assigned as a second argument to the withColumn() function. Note that the second argument should be `Column` type . Also, see Different Ways to Update PySpark DataFrame Column.

```
df.withColumn("salary",col("salary")*100).show()
```

Copy

This snippet multiplies the value of "salary" with 100 and updates the value back to "salary" column.

3. Create a Column from an Existing

To add/create a new column, specify the first argument with a name you want your new column to be and use the second argument to assign a value by applying an operation on an existing column. Also, see Different Ways to Add New Column to PySpark DataFrame.

```
df.withColumn("CopiedColumn",col("salary")* -1).show()
```

Copy

This snippet creates a new column "CopiedColumn" by multiplying "salary" column with value -1.

4. Add a New Column using withColumn()

In order to create a new column, pass the column name you wanted to the first argument of `withColumn()` transformation function. Make sure this new column not already present on DataFrame, if it presents it updates the value of that column.

On below snippet, `PySpark lit()` function is used to add a constant value to a DataFrame column. We can also chain in order to add multiple columns.

```
df.withColumn("Country", lit("USA")).show()
df.withColumn("Country", lit("USA")) \
  .withColumn("anotherColumn",lit("anotherValue")) \
  .show()
```

Copy

5. Rename Column Name

Though you cannot rename a column using withColumn, still I wanted to cover this as renaming is one of the common operations we perform on DataFrame. To rename an existing column use `withColumnRenamed()` function on DataFrame.

```
df.withColumnRenamed("gender","sex") \
  .show(truncate=False)
```

Copy

6. Drop Column From PySpark DataFrame

Use “drop” function to drop a specific column from the DataFrame.

```
df.drop("salary") \
  .show()
```

Copy

Note: Note that all of these functions return the new DataFrame after applying the functions instead of updating DataFrame.

7. PySpark withColumn() Complete Example

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit
from pyspark.sql.types import StructType, StructField, StringType,IntegerType

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

data = [('James','','Smith','1991-04-01','M',3000),
  ('Michael','Rose','','2000-05-19','M',4000),
  ('Robert','','Williams','1978-09-05','M',4000),
  ('Maria','Anne','Jones','1967-12-01','F',4000),
  ('Jen','Mary','Brown','1980-02-17','F',-1)
]

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

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

df2 = df.withColumn("salary",col("salary").cast("Integer"))
df2.printSchema()
df2.show(truncate=False)

df3 = df.withColumn("salary",col("salary")*100)
df3.printSchema()
df3.show(truncate=False)

df4 = df.withColumn("CopiedColumn",col("salary")* -1)
df4.printSchema()

df5 = df.withColumn("Country", lit("USA"))
df5.printSchema()

df6 = df.withColumn("Country", lit("USA")) \
   .withColumn("anotherColumn",lit("anotherValue"))
df6.printSchema()

df.withColumnRenamed("gender","sex") \
  .show(truncate=False)

df4.drop("CopiedColumn") \
.show(truncate=False)
```

PySpark Where Filter Function | Multiple Conditions

- Post author:
- NNK
- Post category:
- PySpark

Table of Contents

PySpark `filter()` function is used to filter the rows from RDD/DataFrame based on the given condition or SQL expression, you can also use `where()` clause instead of the filter() if you are coming from an SQL background, both these functions operate exactly the same.

In this PySpark article, you will learn how to apply a filter on DataFrame columns of string, arrays, struct types by using single and multiple conditions and also applying filter using `isin()` with PySpark (Python Spark) examples.

Related Article:

- How to Filter Rows with NULL/NONE (IS NULL & IS NOT NULL) in PySpark
- Spark Filter – startsWith(), endsWith() Examples
- Spark Filter – contains(), like(), rlike() Examples

Note: PySpark Column Functions provides several options that can be used with filter().

1. PySpark DataFrame filter() Syntax

Below is syntax of the filter function. condition would be an expression you wanted to filter.

```
filter(condition)
```

Copy

Before we start with examples, first let's create a DataFrame. Here, I am using a DataFrame with StructType and ArrayType columns as I will also be covering examples with struct and array types as-well.

```
from pyspark.sql.types import StructType,StructField
from pyspark.sql.types import StringType, IntegerType, ArrayType
data = [
    (("James","","Smith"),["Java","Scala","C++"],"OH","M"),
    (("Anna","Rose",""),["Spark","Java","C++"],"NY","F"),
    (("Julia","","Williams"),["CSharp","VB"],"OH","F"),
    (("Maria","Anne","Jones"),["CSharp","VB"],"NY","M"),
    (("Jen","Mary","Brown"),["CSharp","VB"],"NY","M"),
    (("Mike","Mary","Williams"),["Python","VB"],"OH","M")
 ]

schema = StructType([
     StructField('name', StructType([
        StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
         StructField('lastname', StringType(), True)
     ])),
     StructField('languages', ArrayType(StringType()), True),
     StructField('state', StringType(), True),
     StructField('gender', StringType(), True)
 ])

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

Copy

This yields below schema and DataFrame results.

```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- languages: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- state: string (nullable = true)
 |-- gender: string (nullable = true)
```

```
+----------------------+------------------+-----+------+
|name                  |languages         |state|gender|
+----------------------+------------------+-----+------+
|[James, , Smith]      |[Java, Scala, C++]|OH   |M     |
|[Anna, Rose, ]        |[Spark, Java, C++]|NY   |F     |
|[Julia, , Williams]   |[CSharp, VB]      |OH   |F     |
|[Maria, Anne, Jones]  |[CSharp, VB]      |NY   |M     |
|[Jen, Mary, Brown]    |[CSharp, VB]      |NY   |M     |
|[Mike, Mary, Williams]|[Python, VB]      |OH   |M     |
+----------------------+------------------+-----+------+
```

Copy

2. DataFrame filter() with Column Condition

Use Column with the condition to filter the rows from DataFrame, using this you can express complex condition by referring column names using `dfObject.colname`

```
# Using equals condition
df.filter(df.state == "OH").show(truncate=False)

+----------------------+------------------+-----+------+
|name                  |languages         |state|gender|
+----------------------+------------------+-----+------+
|[James, , Smith]      |[Java, Scala, C++]|OH   |M     |
|[Julia, , Williams]   |[CSharp, VB]      |OH   |F     |
|[Mike, Mary, Williams]|[Python, VB]      |OH   |M     |
+----------------------+------------------+-----+------+

# not equals condition
df.filter(df.state != "OH") \
    .show(truncate=False)
df.filter(~(df.state == "OH")) \
    .show(truncate=False)
```

Copy

Same example can also written as below. In order to use this first you need to import `from pyspark.sql.functions import col`

```
#Using SQL col() function
from pyspark.sql.functions import col
df.filter(col("state") == "OH") \
    .show(truncate=False)
```

Copy

3. DataFrame filter() with SQL Expression

If you are coming from SQL background, you can use that knowledge in PySpark to filter DataFrame rows with SQL expressions.

```
#Using SQL Expression
df.filter("gender == 'M'").show()
#For not equal
df.filter("gender != 'M'").show()
df.filter("gender <> 'M'").show()
```

Copy

4. PySpark Filter with Multiple Conditions

In PySpark, to `filter()` rows on DataFrame based on multiple conditions, you case use either `Column` with a condition or SQL expression. Below is just a simple example using

AND (&) condition, you can extend this with OR(|), and NOT(!) conditional expressions as needed.

```
//Filter multiple condition
df.filter( (df.state  == "OH") & (df.gender  == "M") ) \
    .show(truncate=False)
```

Copy

This yields below DataFrame results.

```
+----------------------+------------------+-----+------+
|name                  |languages         |state|gender|
+----------------------+------------------+-----+------+
|[James, , Smith]      |[Java, Scala, C++]|OH   |M     |
|[Mike, Mary, Williams]|[Python, VB]      |OH   |M     |
+----------------------+------------------+-----+------+
```

Copy

5. Filter Based on List Values

If you have a list of elements and you wanted to filter that is not in the list or in the list, use `isin()` function of `Column` class and it doesn't have isnotin() function but you do the same using not operator (~)

```
#Filter IS IN List values
li=["OH","CA","DE"]
df.filter(df.state.isin(li)).show()
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|    [James, , Smith]|[Java, Scala, C++]|   OH|     M|
| [Julia, , Williams]|      [CSharp, VB]|   OH|     F|
|[Mike, Mary, Will...|      [Python, VB]|   OH|     M|
+--------------------+------------------+-----+------+

# Filter NOT IS IN List values
#These show all records with NY (NY is not part of the list)
df.filter(~df.state.isin(li)).show()
df.filter(df.state.isin(li)==False).show()
```

Copy

6. Filter Based on Starts With, Ends With, Contains

You can also filter DataFrame rows by using `startswith()`, `endswith()` and `contains()` methods of Column class. For more examples on Column class, refer to PySpark Column Functions.

```
# Using startswith
df.filter(df.state.startswith("N")).show()
+--------------------+------------------+-----+------+
|                name|         languages|state|gender|
+--------------------+------------------+-----+------+
|      [Anna, Rose, ]|[Spark, Java, C++]|   NY|     F|
|[Maria, Anne, Jones]|      [CSharp, VB]|   NY|     M|
|  [Jen, Mary, Brown]|      [CSharp, VB]|   NY|     M|
+--------------------+------------------+-----+------+

#using endswith
df.filter(df.state.endswith("H")).show()

#contains
df.filter(df.state.contains("H")).show()
```

Copy

7. PySpark Filter like and rlike

If you have SQL background you must be familiar with `like` and `rlike` (regex like), PySpark also provides similar methods in Column class to filter similar values using wildcard characters. You can use rlike() to filter by checking values case insensitive.

```
data2 = [(2,"Michael Rose"),(3,"Robert Williams"),
     (4,"Rames Rose"),(5,"Rames rose")
  ]
df2 = spark.createDataFrame(data = data2, schema = ["id","name"])

# like - SQL LIKE pattern
df2.filter(df2.name.like("%rose%")).show()
+---+----------+
| id|      name|
+---+----------+
|  5|Rames rose|
+---+----------+

# rlike - SQL RLIKE pattern (LIKE with Regex)
#This check case insensitive
df2.filter(df2.name.rlike("(?i)^*rose$")).show()
+---+------------+
| id|        name|
+---+------------+
|  2|Michael Rose|
|  4|  Rames Rose|
|  5|  Rames rose|
```

Copy

8. Filter on an Array column

When you want to filter rows from DataFrame based on value present in an array collection column, you can use the first syntax. The below example uses `array_contains()` from Pyspark SQL functions which checks if a value contains in an array if present it returns true otherwise false.

```
from pyspark.sql.functions import array_contains
df.filter(array_contains(df.languages,"Java")) \
    .show(truncate=False)
```

Copy

This yields below DataFrame results.

```
+----------------+------------------+-----+------+
|name            |languages         |state|gender|
+----------------+------------------+-----+------+
|[James, , Smith]|[Java, Scala, C++]|OH   |M     |
|[Anna, Rose, ]  |[Spark, Java, C++]|NY   |F     |
+----------------+------------------+-----+------+
```

Copy

9. Filtering on Nested Struct columns

If your DataFrame consists of nested struct columns, you can use any of the above syntaxes to filter the rows based on the nested column.

```
  //Struct condition
df.filter(df.name.lastname == "Williams") \
    .show(truncate=False)
```

Copy

This yields below DataFrame results

```
+----------------------+------------+-----+------+
|name                  |languages   |state|gender|
+----------------------+------------+-----+------+
|[Julia, , Williams]   |[CSharp, VB]|OH   |F     |
|[Mike, Mary, Williams]|[Python, VB]|OH   |M     |
+----------------------+------------+-----+------+
```

Copy

10. Source code of PySpark where filter

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField, StringType, IntegerType, ArrayType
from pyspark.sql.functions import col,array_contains

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

arrayStructureData = [
        (("James","","Smith"),["Java","Scala","C++"],"OH","M"),
        (("Anna","Rose",""),["Spark","Java","C++"],"NY","F"),
        (("Julia","","Williams"),["CSharp","VB"],"OH","F"),
        (("Maria","Anne","Jones"),["CSharp","VB"],"NY","M"),
        (("Jen","Mary","Brown"),["CSharp","VB"],"NY","M"),
        (("Mike","Mary","Williams"),["Python","VB"],"OH","M")
        ]

arrayStructureSchema = StructType([
        StructField('name', StructType([
             StructField('firstname', StringType(), True),
            StructField('middlename', StringType(), True),
            StructField('lastname', StringType(), True)
            ])),
         StructField('languages', ArrayType(StringType()), True),
         StructField('state', StringType(), True),
         StructField('gender', StringType(), True)
         ])


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

df.filter(df.state == "OH") \
    .show(truncate=False)

df.filter(col("state") == "OH") \
    .show(truncate=False)

df.filter("gender  == 'M'") \
    .show(truncate=False)

df.filter( (df.state  == "OH") & (df.gender  == "M") ) \
    .show(truncate=False)

df.filter(array_contains(df.languages,"Java")) \
    .show(truncate=False)

df.filter(df.name.lastname == "Williams") \
    .show(truncate=False)
```

PySpark Select Columns From DataFrame

- Post author:
- NNK
- Post category:
- PySpark

Table of Contents

In PySpark, `select()` function is used to select single, multiple, column by index, all columns from the list and the nested columns from a DataFrame, PySpark select() is a transformation function hence it returns a new DataFrame with the selected columns.

- Select a Single & Multiple Columns from PySpark
- Select All Columns From List
- Select Columns By Index
- Select a Nested Column
- Other Ways to Select Columns

First, let's create a Dataframe.

```
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)
```

Copy

1. Select Single & Multiple Columns From PySpark

You can select the single or multiple columns of the DataFrame by passing the column names you wanted to select to the `select()` function. Since DataFrame is immutable, this creates a new DataFrame with selected columns. show() function is used to show the Dataframe contents.

Below are ways to select single, multiple or all columns.

```
df.select("firstname","lastname").show()
df.select(df.firstname,df.lastname).show()
df.select(df["firstname"],df["lastname"]).show()

#By using col() function
from pyspark.sql.functions import col
df.select(col("firstname"),col("lastname")).show()

#Select columns by regular expression
df.select(df.colRegex("`^.*name*`")).show()
```

Copy

2. Select All Columns From List

Sometimes you may need to select all DataFrame columns from a Python list. In the below example, we have all columns in the `columns` list object.

```
# Select All columns from List
df.select(*columns).show()

# Select All columns
df.select([col for col in df.columns]).show()
df.select("*").show()
```

Copy

3. Select Columns by Index

Using a python list features, you can select the columns by index.

```
#Selects first 3 columns and top 3 rows
df.select(df.columns[:3]).show(3)

#Selects columns 2 to 4  and top 3 rows
df.select(df.columns[2:4]).show(3)
```

Copy

4. Select Nested Struct Columns from PySpark

If you have a nested struct (StructType) column on PySpark DataFrame, you need to use an explicit column qualifier in order to select. If you are new to PySpark and you have not learned StructType yet, I would recommend skipping the rest of the section or first Understand PySpark StructType before you proceed.

First, let’s create a new DataFrame with a struct type.

```
data = [
        (("James",None,"Smith"),"OH","M"),
        (("Anna","Rose",""),"NY","F"),
        (("Julia","","Williams"),"OH","F"),
        (("Maria","Anne","Jones"),"NY","M"),
        (("Jen","Mary","Brown"),"NY","M"),
        (("Mike","Mary","Williams"),"OH","M")
        ]

from pyspark.sql.types import StructType,StructField, StringType
```

```
schema = StructType([
    StructField('name', StructType([
         StructField('firstname', StringType(), True),
         StructField('middlename', StringType(), True),
         StructField('lastname', StringType(), True)
         ])),
     StructField('state', StringType(), True),
     StructField('gender', StringType(), True)
     ])
df2 = spark.createDataFrame(data = data, schema = schema)
df2.printSchema()
df2.show(truncate=False) # shows all columns
```

Copy

Yields below schema output. If you notice the `column` `name` is a `struct` type which consists of columns `firstname`, `middlename`, `lastname`.

```
root
 |-- name: struct (nullable = true)
 |    |-- firstname: string (nullable = true)
 |    |-- middlename: string (nullable = true)
 |    |-- lastname: string (nullable = true)
 |-- state: string (nullable = true)
 |-- gender: string (nullable = true)

+----------------------+-----+------+
|name                  |state|gender|
+----------------------+-----+------+
|[James,, Smith]       |OH   |M     |
|[Anna, Rose, ]        |NY   |F     |
|[Julia, , Williams]   |OH   |F     |
|[Maria, Anne, Jones]  |NY   |M     |
|[Jen, Mary, Brown]    |NY   |M     |
|[Mike, Mary, Williams]|OH   |M     |
+----------------------+-----+------+
```

Copy

Now, let's select struct column.

```
df2.select("name").show(truncate=False)
```

Copy

This returns struct column `name` as is.

```
+----------------------+
|name                  |
+----------------------+
|[James,, Smith]       |
|[Anna, Rose, ]        |
|[Julia, , Williams]   |
|[Maria, Anne, Jones]  |
|[Jen, Mary, Brown]    |
|[Mike, Mary, Williams]|
+----------------------+
```

Copy

In order to select the specific column from a nested struct, you need to explicitly qualify the nested struct column name.

```
df2.select("name.firstname","name.lastname").show(truncate=False)
```

Copy

This outputs `firstname` and `lastname` from the name struct column.

```
+---------+--------+
|firstname|lastname|
+---------+--------+
|James    |Smith   |
|Anna     |        |
|Julia    |Williams|
|Maria    |Jones   |
|Jen      |Brown   |
|Mike     |Williams|
+---------+--------+
```

Copy

In order to get all columns from struct column.

```
df2.select("name.*").show(truncate=False)
```

Copy

This yields below output.

```
+---------+----------+--------+
|firstname|middlename|lastname|
+---------+----------+--------+
|James    |null      |Smith   |
|Anna     |Rose      |        |
|Julia    |          |Williams|
|Maria    |Anne      |Jones   |
|Jen      |Mary      |Brown   |
|Mike     |Mary      |Williams|
+---------+----------+--------+
```

Copy

5. Complete Example

```
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)

df.select("firstname").show()

df.select("firstname","lastname").show()

#Using Dataframe object name
df.select(df.firstname,df.lastname).show()

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

data = [(("James",None,"Smith"),"OH","M"),
        (("Anna","Rose",""),"NY","F"),
        (("Julia","","Williams"),"OH","F"),
        (("Maria","Anne","Jones"),"NY","M"),
        (("Jen","Mary","Brown"),"NY","M"),
        (("Mike","Mary","Williams"),"OH","M")
        ]

from pyspark.sql.types import StructType,StructField, StringType
schema = StructType([
    StructField('name', StructType([
         StructField('firstname', StringType(), True),
        StructField('middlename', StringType(), True),
        StructField('lastname', StringType(), True)
        ])),
    StructField('state', StringType(), True),
    StructField('gender', StringType(), True)
    ])

df2 = spark.createDataFrame(data = data, schema = schema)
df2.printSchema()
df2.show(truncate=False) # shows all columns

df2.select("name").show(truncate=False)
df2.select("name.firstname","name.lastname").show(truncate=False)
df2.select("name.*").show(truncate=False)
```

Copy

This example is also available at PySpark github project.

PySpark orderBy() and sort() explained

- Post author:
- Admin
- Post category:
- PySpark

Table of Contents

You can use either `sort()` or `orderBy()` function of PySpark DataFrame to sort DataFrame by ascending or descending order based on single or multiple columns, you can also do sorting using PySpark SQL sorting functions, In this article, I will explain all these different ways using PySpark examples.

- Using sort() function
- Using orderBy() function
- Ascending order
- Descending order
- SQL Sort functions

Related: How to sort DataFrame by using Scala

Before we start, first let's create a DataFrame.

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

Copy

This Yields below output.

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

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

Copy

DataFrame sorting using the sort() function

PySpark DataFrame class provides `sort()` function to sort on one or more columns. By default, it sorts by ascending order.

Syntax

```
sort(self, *cols, **kwargs):
```

Copy

Example

```
df.sort("department","state").show(truncate=False)
df.sort(col("department"),col("state")).show(truncate=False)
```

Copy

The above two examples return the same below output, the first one takes the DataFrame column name as a string and the next takes columns in Column type. This table sorted by the first `department` column and then the `state` column.

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|Maria        |Finance   |CA   |90000 |24 |23000|
|Raman        |Finance   |CA   |99000 |40 |24000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Scott        |Finance   |NY   |83000 |36 |19000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
+-------------+----------+-----+------+---+-----+
```

Copy

DataFrame sorting using orderBy() function

PySpark DataFrame also provides `orderBy()` function to sort on one or more columns. By default, it orders by ascending.

Example

```
df.orderBy("department","state").show(truncate=False)
df.orderBy(col("department"),col("state")).show(truncate=False)
```

Copy

This returns the same output as the previous section.

Sort by Ascending (ASC)

If you wanted to specify the ascending order/sort explicitly on DataFrame, you can use the `asc` method of the `Column` function. for example

```
df.sort(df.department.asc(),df.state.asc()).show(truncate=False)
df.sort(col("department").asc(),col("state").asc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").asc()).show(truncate=False)
```

Copy

The above three examples return the same output.

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|Maria        |Finance   |CA   |90000 |24 |23000|
|Raman        |Finance   |CA   |99000 |40 |24000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Scott        |Finance   |NY   |83000 |36 |19000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
|Robert       |Sales     |CA   |81000 |30 |23000|
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
+-------------+----------+-----+------+---+-----+
```

Copy

Sort by Descending (DESC)

If you wanted to specify the sorting by descending order on DataFrame, you can use the `desc` method of the `Column` function. for example. From our example, let’s use desc on the state column.

```
df.sort(df.department.asc(),df.state.desc()).show(truncate=False)
df.sort(col("department").asc(),col("state").desc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").desc()).show(truncate=False)
```

Copy

This yields the below output for all three examples.

```
+-------------+----------+-----+------+---+-----+
|employee_name|department|state|salary|age|bonus|
+-------------+----------+-----+------+---+-----+
|Scott        |Finance   |NY   |83000 |36 |19000|
|Jen          |Finance   |NY   |79000 |53 |15000|
|Raman        |Finance   |CA   |99000 |40 |24000|
|Maria        |Finance   |CA   |90000 |24 |23000|
|Kumar        |Marketing |NY   |91000 |50 |21000|
|Jeff         |Marketing |CA   |80000 |25 |18000|
|James        |Sales     |NY   |90000 |34 |10000|
|Michael      |Sales     |NY   |86000 |56 |20000|
|Robert       |Sales     |CA   |81000 |30 |23000|
+-------------+----------+-----+------+---+-----+
```

Copy

Besides `asc()` and `desc()` functions, PySpark also provides `asc_nulls_first()` and `asc_nulls_last()` and equivalent descending functions.

Using Raw SQL

Below is an example of how to sort DataFrame using raw SQL syntax.

```
df.createOrReplaceTempView("EMP")
spark.sql("select employee_name,department,state,salary,age,bonus from EMP ORDER BY
department asc").show(truncate=False)
```

Copy

The above two examples return the same output as above.

Dataframe Sorting Complete Example

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, asc,desc

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

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) \
  ]
columns= ["employee_name","department","state","salary","age","bonus"]

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

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

df.sort("department","state").show(truncate=False)
df.sort(col("department"),col("state")).show(truncate=False)

df.orderBy("department","state").show(truncate=False)
df.orderBy(col("department"),col("state")).show(truncate=False)

df.sort(df.department.asc(),df.state.asc()).show(truncate=False)
df.sort(col("department").asc(),col("state").asc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").asc()).show(truncate=False)

df.sort(df.department.asc(),df.state.desc()).show(truncate=False)
df.sort(col("department").asc(),col("state").desc()).show(truncate=False)
df.orderBy(col("department").asc(),col("state").desc()).show(truncate=False)

df.createOrReplaceTempView("EMP")
spark.sql("select employee_name,department,state,salary,age,bonus from EMP ORDER BY 
department asc").show(truncate=False)
```

Copy

PySpark explode array and map columns to rows

- Post author:
- NNK
- Post category:
- PySpark

Table of Contents

In this article, I will explain how to explode array or list and map columns to rows using different PySpark DataFrame functions (explode, explore_outer, posexplode, posexplode_outer) with Python example.

Before we start, let's create a DataFrame with array and map fields, below snippet, creates a DF with columns "name" as StringType, "knownLanguage" as ArrayType and "properties" as MapType.

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('pyspark-by-examples').getOrCreate()

arrayData = [
        ('James',['Java','Scala'],{'hair':'black','eye':'brown'}),
        ('Michael',['Spark','Java',None],{'hair':'brown','eye':None}),
        ('Robert',['CSharp',''],{'hair':'red','eye':''}),
        ('Washington',None,None),
        ('Jefferson',['1','2'],{})

df = spark.createDataFrame(data=arrayData, schema = ['name','knownLanguages','properties'])
df.printSchema()
df.show()
```

Copy

Outputs:

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

```
+----------+--------------+--------------------+
|      name|knownLanguages|          properties|
+----------+--------------+--------------------+
|     James| [Java, Scala]|[eye -> brown, ha...|
|   Michael|[Spark, Java,]|[eye ->, hair -> ...|
|    Robert|    [CSharp, ]|[eye -> , hair ->...|
|Washington|          null|                null|
| Jefferson|        [1, 2]|                  []|
+----------+--------------+--------------------+
```

Copy

explode – PySpark explode array or map column to rows

PySpark function `explode(e: Column)` is used to explode or create array or map columns to rows. When an array is passed to this function, it creates a new default column "col1" and it contains all array elements. When a map is passed, it creates two new columns one for key and one for value and each element in map split into the rows.

This will ignore elements that have null or empty. from the above example, Washington and Jefferson have null or empty values in array and map, hence the following snippet out does not contain these rows.

explode – array column example

```
from pyspark.sql.functions import explode
df2 = df.select(df.name,explode(df.knownLanguages))
df2.printSchema()
df2.show()
```

Copy

Outputs

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

+---------+------+
|     name|   col|
+---------+------+
|    James|  Java|
|    James| Scala|
|  Michael| Spark|
|  Michael|  Java|
|  Michael|  null|
|   Robert|CSharp|
|   Robert|      |
|Jefferson|     1|
|Jefferson|     2|
+---------+------+
```

Copy

explode – map column example

```
from pyspark.sql.functions import explode
df3 = df.select(df.name,explode(df.properties))
df3.printSchema()
df3.show()
```

Copy

Outputs:

```
root
 |-- name: string (nullable = true)
 |-- key: string (nullable = false)
 |-- value: string (nullable = true)

+-------+----+-----+
|   name| key|value|
+-------+----+-----+
|  James| eye|brown|
|  James|hair|black|
|Michael| eye| null|
|Michael|hair|brown|
| Robert| eye|     |
| Robert|hair|  red|
+-------+----+-----+
```

Copy

explode_outer – Create rows for each element in an array or map.

PySpark SQL `explode_outer(e: Column)` function is used to create a row for each element in the array or map column. Unlike explode, if the array or map is null or empty, explode_outer returns null.

```
from pyspark.sql.functions import explode_outer
""" with array """
df.select(df.name,explode_outer(df.knownLanguages)).show()
""" with map """
df.select(df.name,explode_outer(df.properties)).show()
```

Copy

posexplode – explode array or map elements to rows

`posexplode(e: Column)` creates a row for each element in the array and creates two columns “pos’ to hold the position of the array element and the ‘col’ to hold the actual array value. And when the input column is a map, posexplode function creates 3 columns “pos” to hold the position of the map element, “key” and “value” columns.

This will ignore elements that have null or empty. Since the Washington and Jefferson have null or empty values in array and map, the following snippet out does not contain these.

```
from pyspark.sql.functions import posexplode
""" with array """
df.select(df.name,posexplode(df.knownLanguages)).show()
""" with map """
df.select(df.name,posexplode(df.properties)).show()
```

Copy

posexplode_outer – explode array or map columns to rows.

Spark `posexplode_outer(e: Column)` creates a row for each element in the array and creates two columns “pos’ to hold the position of the array element and the ‘col’ to hold the actual array value. Unlike posexplode, if the array or map is null or empty, posexplode_outer function returns null, null for pos and col columns. Similarly for the map, it returns rows with nulls.

```
from pyspark.sql.functions import posexplode_outer
""" with array """
df.select($"name",posexplode_outer($"knownLanguages")).show()

""" with map """
df.select(df.name,posexplode_outer(df.properties)).show()
```

PySpark – explode nested array into rows

- Post author:
- NNK
- Post category:
- PySpark

Problem: How to explode & flatten nested array (Array of Array) DataFrame columns into rows using PySpark.

Solution: PySpark explode function can be used to explode an Array of Array (nested Array) `ArrayType(ArrayType(StringType))` columns to rows on PySpark DataFrame using python example.

Before we start, let’s create a DataFrame with a nested array column. From below example column “subjects” is an array of ArraType which holds subjects learned.

```
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)
```

Copy

df.printSchema() and df.show() returns the following schema and table.

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

Copy

Now, let’s explode “subjects” array column to array rows. after exploding, it creates a new column ‘col’ with rows represents an array.

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

Copy

Outputs:

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

Copy

If you want to flatten the arrays, use flatten function which converts array of array columns to a single array on DataFrame.

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

Copy

Outputs:

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

PySpark Groupby Explained with Example

- Post author:
- NNK
- Post category:
- PySpark

Table of Contents

Similar to SQL `GROUP BY` clause, PySpark `groupBy()` function is used to collect the identical data into groups on DataFrame and perform aggregate functions on the grouped data. In this article, I will explain several `groupBy()` examples using PySpark (Spark with Python).

Related: [How to group and aggregate data using Spark and Scala]()

Syntax:

```
groupBy(col1 : scala.Predef.String, cols : scala.Predef.String*) :
      org.apache.spark.sql.RelationalGroupedDataset
```

Copy

When we perform `groupBy()` on PySpark Dataframe, it returns `GroupedData` object which contains below aggregate functions.

`count()` – Returns the count of rows for each group.

`mean()` – Returns the mean of values for each group.

`max()` – Returns the maximum of values for each group.

`min()` – Returns the minimum of values for each group.

`sum()` – Returns the total for values for each group.

`avg()` – Returns the average for values for each group.

`agg()` – Using agg() function, we can calculate more than one aggregate at a time.

`pivot()` – This function is used to Pivot the DataFrame which I will not be covered in this article as I already have a dedicated article for Pivot & Unpivot DataFrame.

Preparing Data & creating DataFrame

Before we start, let's create the DataFrame from a sequence of the data to work with. This DataFrame contains columns "`employee_name`", "`department`", "`state`", "`salary`", "`age`" and "`bonus`" columns.

We will use this PySpark DataFrame to run groupBy() on "department" columns and calculate aggregates like minimum, maximum, average, total salary for each group using min(), max() and sum() aggregate functions respectively. and finally, we will also see how to do group and aggregate on multiple columns.

```
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)
```

Copy

Yields below output.

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

Copy

PySpark groupBy and aggregate on DataFrame columns

Let’s do the `groupBy()` on `department` column of DataFrame and then find the sum of salary for each department using `sum()` aggregate function.

```
df.groupBy("department").sum("salary").show(truncate=False)
+----------+-----------+
|department|sum(salary)|
+----------+-----------+
|Sales     |257000     |
|Finance   |351000     |
|Marketing |171000     |
+----------+-----------+
```

Copy

Similarly, we can calculate the number of employee in each department using `count()`

```
df.groupBy("department").count()
```

Copy

Calculate the minimum salary of each department using `min()`

```
df.groupBy("department").min("salary")
```

Copy

Calculate the maximin salary of each department using `max()`

```
df.groupBy("department").max("salary")
```

Copy

Calculate the average salary of each department using `avg()`

```
df.groupBy("department").avg( "salary")
```

Copy

Calculate the mean salary of each department using `mean()`

```
df.groupBy("department").mean( "salary")
```

Copy

PySpark groupBy and aggregate on multiple columns

Similarly, we can also run groupBy and aggregate on two or more DataFrame columns, below example does group by on `department`,`state` and does sum() on `salary` and `bonus` columns.

```
//GroupBy on multiple columns
df.groupBy("department","state") \
    .sum("salary","bonus") \
    .show(false)
```

Copy

This yields the below output.

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

Copy

similarly, we can run group by and aggregate on tow or more columns for other aggregate functions, please refer below source code for example.

Running more aggregates at a time

Using `agg()` aggregate function we can calculate many aggregations at a time on a single statement using PySpark SQL aggregate functions sum(), avg(), min(), max() mean() e.t.c. In order to use these, we should import `"from pyspark.sql.functions import sum,avg,max,min,mean,count"`

```
df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
         avg("salary").alias("avg_salary"), \
         sum("bonus").alias("sum_bonus"), \
         max("bonus").alias("max_bonus") \
     ) \
    .show(truncate=False)
```

Copy

This example does group on `department` column and calculates `sum()` and `avg()` of `salary` for each department and calculates `sum()` and `max()` of bonus for each department.

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

Copy

Using filter on aggregate data

Similar to SQL "HAVING" clause, On PySpark DataFrame we can use either where() or filter() function to filter the rows of aggregated data.

```
df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
      avg("salary").alias("avg_salary"), \
      sum("bonus").alias("sum_bonus"), \
      max("bonus").alias("max_bonus")) \
    .where(col("sum_bonus") >= 50000) \
    .show(truncate=False)
```

Copy

This removes the sum of a bonus that has less than 50000 and yields below output.

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

Copy

PySpark groupBy Example Source code

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col,sum,avg,max

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

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)

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

df.groupBy("department").count().show(truncate=False)
df.groupBy("department","state") \
    .sum("salary","bonus") \
   .show(truncate=False)

df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
         avg("salary").alias("avg_salary"), \
         sum("bonus").alias("sum_bonus"), \
         max("bonus").alias("max_bonus") \
     ) \
    .show(truncate=False)

df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
      avg("salary").alias("avg_salary"), \
      sum("bonus").alias("sum_bonus"), \
      max("bonus").alias("max_bonus")) \
    .where(col("sum_bonus") >= 50000) \
    .show(truncate=False)
```

PySpark Aggregate Functions with Examples

- Post author:
- NNK
- Post category:
- PySpark

Table of Contents

PySpark provides built-in standard Aggregate functions defines in DataFrame API, these come in handy when we need to make aggregate operations on DataFrame columns. Aggregate functions operate on a group of rows and calculate a single return value for every group.

All these aggregate functions accept input as, Column type or column name in a string and several other arguments based on the function and return Column type.

When possible try to leverage standard library as they are little bit more compile-time safety, handles null and perform better when compared to UDF's. If your application is critical on performance try to avoid using custom UDF at all costs as these are not guarantee on performance.

PySpark Aggregate Functions

PySpark SQL Aggregate functions are grouped as "agg_funcs" in Pyspark. Below is a list of functions defined under this group. Click on each link to learn with example.

- approx_count_distinct
- avg
- collect_list
- collect_set
- countDistinct
- count
- grouping
- first
- last
- kurtosis
- max
- min
- mean
- skewness
- stddev
- stddev_samp
- stddev_pop
- sum
- sumDistinct
- variance, var_samp, var_pop

PySpark Aggregate Functions Examples

First, let’s create a DataFrame to work with PySpark aggregate functions. All examples provided here are also available at PySpark Examples GitHub project.

```
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)
```

Copy

Yields below output.

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

Copy

Now let’s see how to aggregate data in PySpark.

approx_count_distinct Aggregate Function

In PySpark `approx_count_distinct()` function returns the count of distinct items in a group.

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

//Prints approx_count_distinct: 6
```

Copy

avg (average) Aggregate Function

`avg()` function returns the average of values in the input column.

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

//Prints avg: 3400.0
```

Copy

collect_list Aggregate Function

`collect_list()` function returns all values from an input column with duplicates.

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

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

Copy

collect_set Aggregate Function

`collect_set()` function returns all values from an input column with duplicate values eliminated.

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

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

Copy

countDistinct Aggregate Function

`countDistinct()` function returns the number of distinct elements in a columns

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

Copy

count function

`count()` function returns number of elements in a column.

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

Prints county: 10
```

Copy

grouping function

`grouping()` Indicates whether a given input column is aggregated or not. returns 1 for aggregated or 0 for not aggregated in the result. If you try grouping directly on the salary column you will get below error.

```
Exception in thread "main" org.apache.spark.sql.AnalysisException:
  // grouping() can only be used with GroupingSets/Cube/Rollup
```

Copy

first function

`first()` function returns the first element in a column when ignoreNulls is set to true, it returns the first non-null element.

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

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

Copy

last function

`last()` function returns the last element in a column. when ignoreNulls is set to true, it returns the last non-null element.

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

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

Copy

kurtosis function

`kurtosis()` function returns the kurtosis of the values in a group.

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

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

Copy

max function

`max()` function returns the maximum value in a column.

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

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

Copy

min function

`min()` function

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

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

Copy

mean function

`mean()` function returns the average of the values in a column. Alias for Avg

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

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

Copy

skewness function

`skewness()` function returns the skewness of the values in a group.

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

+--------------------+
|skewness(salary)    |
+--------------------+
|-0.12041791181069571|
+--------------------+
```

Copy

stddev(), stddev_samp() and stddev_pop()

`stddev()` alias for `stddev_samp`.

`stddev_samp()` function returns the sample standard deviation of values in a column.

`stddev_pop()` function returns the population standard deviation of the values in a column.

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

Copy

sum function

`sum()` function Returns the sum of all values in a column.

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

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

Copy

sumDistinct function

`sumDistinct()` function returns the sum of all distinct values in a column.

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

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

Copy

variance(), var_samp(), var_pop()

`variance()` alias for `var_samp`

`var_samp()` function returns the unbiased variance of the values in a column.

`var_pop()` function returns the population variance of the values in a column.

```
df.select(variance("salary"),var_samp("salary"),var_pop("salary")) \
  .show(truncate=False)

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

Copy

Source code of PySpark Aggregate examples

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import approx_count_distinct,collect_list
from pyspark.sql.functions import collect_set,sum,avg,max,countDistinct,count
from pyspark.sql.functions import first, last, kurtosis, min, mean, skewness
from pyspark.sql.functions import stddev, stddev_samp, stddev_pop, sumDistinct
from pyspark.sql.functions import variance,var_samp,  var_pop

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

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)

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

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

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

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

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

print("count: "+str(df.select(count("salary")).collect()[0]))
df.select(first("salary")).show(truncate=False)
df.select(last("salary")).show(truncate=False)
df.select(kurtosis("salary")).show(truncate=False)
df.select(max("salary")).show(truncate=False)
df.select(min("salary")).show(truncate=False)
df.select(mean("salary")).show(truncate=False)
df.select(skewness("salary")).show(truncate=False)
df.select(stddev("salary"), stddev_samp("salary"), \
    stddev_pop("salary")).show(truncate=False)
df.select(sum("salary")).show(truncate=False)
df.select(sumDistinct("salary")).show(truncate=False)
df.select(variance("salary"),var_samp("salary"),var_pop("salary")) \
  .show(truncate=False)
```

PySpark Join Types | Join Two DataFrames

- Post author:
- NNK
- Post category:
- PySpark

Table of Contents

PySpark Join is used to combine two DataFrames and by chaining these you can join multiple DataFrames; it supports all basic join type operations available in traditional SQL like `INNER`, `LEFT OUTER`, `RIGHT OUTER`, `LEFT ANTI`, `LEFT SEMI`, `CROSS`, `SELF` JOIN. PySpark Joins are wider transformations that involve data shuffling across the network.

PySpark SQL Joins comes with more optimization by default (thanks to DataFrames) however still there would be some performance issues to consider while using.

In this PySpark SQL Join tutorial, you will learn different Join syntaxes and using different Join types on two or more DataFrames and Datasets using examples.

- PySpark Join Syntax

- PySpark Join Types
- Inner Join DataFrame
- Full Outer Join DataFrame
- Left Outer Join DataFrame
- Right Outer Join DataFrame
- Left Anti Join DataFrame
- Left Semi Join DataFrame
- Self Join DataFrame
- Using SQL Expression

1. PySpark Join Syntax

PySpark SQL join has a below syntax and it can be accessed directly from DataFrame.

```
join(self, other, on=None, how=None)
```

Copy

`join()` operation takes parameters as below and returns DataFrame.

- param other: Right side of the join
- param on: a string for the join column name
- param how: default `inner`. Must be one of `inner`, `cross`, `outer`,`full`, `full_outer`, `left`, `left_outer`, `right`, `right_outer`,`left_semi`, and `left_anti`.

You can also write Join expression by adding where() and filter() methods on DataFrame and can have Join on multiple columns.

2. PySpark Join Types

Below are the different Join Types PySpark supports.

Join String	Equivalent SQL Join
inner	INNER JOIN
outer, full, fullouter, full_outer	FULL OUTER JOIN
left, leftouter, left_outer	LEFT JOIN
right, rightouter, right_outer	RIGHT JOIN
cross	
anti, leftanti, left_anti	
semi, leftsemi, left_semi	

PySpark Join Types

Before we jump into PySpark SQL Join examples, first, let's create an `"emp"` and `"dept"` DataFrames. here, column `"emp_id"` is unique on emp and `"dept_id"` is unique on the dept dataset's and emp_dept_id from emp has a reference to dept_id on dept dataset.

```
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)

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)
```

Copy

This prints "emp" and "dept" DataFrame to the console. Refer complete example below on how to create `spark` object.

Emp Dataset

```
+------+--------+---------------+-----------+-----------+------+------+
|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 Dataset
+---------+-------+
|dept_name|dept_id|
+---------+-------+
|Finance  |10     |
|Marketing|20     |
|Sales    |30     |
|IT       |40     |
+---------+-------+
```

Copy

3. PySpark Inner Join DataFrame

`Inner` join is the default join in PySpark and it's mostly used. This joins two datasets on key columns, where keys don't match the rows get dropped from both datasets (`emp` & `dept`).

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

Copy

When we apply Inner join on our datasets, It drops "`emp_dept_id`" 50 from "`emp`" and "`dept_id`" 30 from "`dept`" datasets. Below is the result of the above Join expression.

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

Copy

4. PySpark Full Outer Join

`Outer` a.k.a `full`, `fullouter` join returns all rows from both datasets, where join expression doesn't match it returns null on respective record columns.

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

Copy

From our "`emp`" dataset's "`emp_dept_id`" with value 50 doesn't have a record on "`dept`" hence dept columns have null and "`dept_id`" 30 doesn't have a record in "`emp`" hence you see null's on emp columns. Below is the result of the above Join expression.

```
+------+--------+---------------+-----------+-----------+------+------+---------+-------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|salary|dept_name|dept_id|
+------+--------+---------------+-----------+-----------+------+------+---------+-------+
|2     |Rose    |1              |2010       |20         |M     |4000  |Marketing|20     |
|5     |Brown   |2              |2010       |40         |      |-1    |IT       |40     |
|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     |
|6     |Brown   |2              |2010       |50         |      |-1    |null     |null   |
|null  |null    |null           |null       |null       |null  |null  |Sales    |30     |
+------+--------+---------------+-----------+-----------+------+------+---------+-------+
```

Copy

5. PySpark Left Outer Join

`Left` a.k.a `Leftouter` join returns all rows from the left dataset regardless of match found on the right dataset when join expression doesn't match, it assigns null for that record and drops records from right where match not found.

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

Copy

From our dataset, "`emp_dept_id`" 5o doesn't have a record on "`dept`" dataset hence, this record contains null on "`dept`" columns (dept_name & dept_id). and "`dept_id`" 30 from "`dept`" dataset dropped from the results. Below is the result of the above Join expression.

```
+------+--------+---------------+-----------+-----------+------+------+---------+-------+
|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     |
|2     |Rose    |1              |2010       |20         |M     |4000  |Marketing|20     |
|3     |Williams|1              |2010       |10         |M     |1000  |Finance  |10     |
|4     |Jones   |2              |2005       |10         |F     |2000  |Finance  |10     |
|5     |Brown   |2              |2010       |40         |      |-1    |IT       |40     |
|6     |Brown   |2              |2010       |50         |      |-1    |null     |null   |
+------+--------+---------------+-----------+-----------+------+------+---------+-------+
```

Copy

6. Right Outer Join

`Right` a.k.a `Rightouter` join is opposite of `left` join, here it returns all rows from the right dataset regardless of math found on the left dataset, when join expression doesn't match, it assigns null for that record and drops records from left where match not found.

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

Copy

From our example, the right dataset "`dept_id`" 30 doesn't have it on the left dataset "`emp`" hence, this record contains null on "`emp`" columns. and "`emp_dept_id`" 50 dropped as a match not found on left. Below is the result of the above Join expression.

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

Copy

7. Left Semi Join

`leftsemi` join is similar to `inner` join difference being `leftsemi` join returns all columns from the left dataset and ignores all columns from the right dataset. In other words, this join returns columns from the only left dataset for the records match in the right dataset on join expression, records not matched on join expression are ignored from both left and right datasets.

The same result can be achieved using select on the result of the inner join however, using this join would be efficient.

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

Copy

Below is the result of the above join expression.

```
leftsemi join
+------+--------+---------------+-----------+-----------+------+------+
|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    |
+------+--------+---------------+-----------+-----------+------+------+
```

Copy

8. Left Anti Join

`leftanti` join does the exact opposite of the `leftsemi`, `leftanti` join returns only columns from the left dataset for non-matched records.

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

Copy

Yields below output

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

Copy

9. PySpark Self Join

Joins are not complete without a self join, Though there is no self-join type available, we can use any of the above-explained join types to join DataFrame to itself. below example use `inner` self join.

```
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)
```

Copy

Here, we are joining `emp` dataset with itself to find out superior `emp_id` and `name` for all employees.

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

Copy

4. Using SQL Expression

Since PySpark SQL support native SQL syntax, we can also write join operations after creating temporary tables on DataFrames and use these tables on `spark.sql()`.

```
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)

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

Copy

5. PySpark SQL Join on multiple DataFrames

When you need to join more than two tables, you either use SQL expression after creating a temporary view on the DataFrame or use the result of join operation to join with another DataFrame like chaining them. for example

```
df1.join(df2,df1.id1 == df2.id2,"inner") \
   .join(df3,df1.id1 == df3.id3,"inner")
```

Copy

6. PySpark SQL Join Complete Example

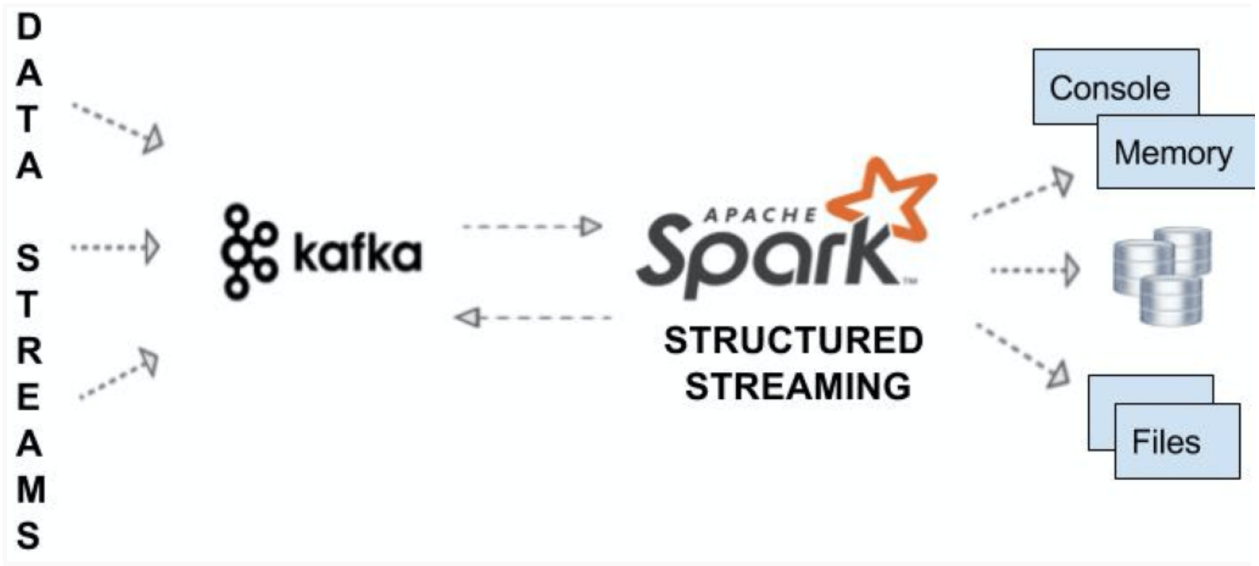
```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

spark = SparkSession.builder.appName('SparkByExamples.com').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)


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)

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

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

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

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

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

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

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)

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)

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

Spark Streaming with Kafka Example

- Post author:
- NNK
- Post category:
- Apache Spark / Apache Spark Streaming

Table of Contents



Spark Streaming with Kafka Example

Using Spark Streaming we can read from Kafka topic and write to Kafka topic in TEXT, CSV, AVRO and JSON formats, In this article, we will learn with scala example of how to stream from Kafka messages in JSON format using `from_json()` and `to_json()` SQL functions.

What is Spark Streaming?

Apache Spark Streaming is a scalable, high-throughput, fault-tolerant streaming processing system that supports both batch and streaming workloads. It is an extension of the core Spark API to process real-time data from sources like Kafka, Flume, and Amazon Kinesis to name a few. This processed data can be pushed to other systems like databases, Kafka, live dashboards e.t.c

What is Apache Kafka

Apache Kafka is a publish-subscribe messaging system originally written at LinkedIn.

A Kafka cluster is a highly scalable and fault-tolerant system and it also has a much higher throughput compared to other message brokers such as ActiveMQ and RabbitMQ

Prerequisites

If you don’t have Kafka cluster setup, follow the below articles to set up the single broker cluster and get familiar with creating and describing topics.

- Install & set-up Kafka Cluster guide
- How to create and describe Kafka topics

Table of contents

- Write JSON to Kafka using producer shell
- Streaming from Kafka topic
- Write Streamed JSON to Console
- Write Streamed JSON to Kafka topic
- Read JSON from Kafka using consumer shell

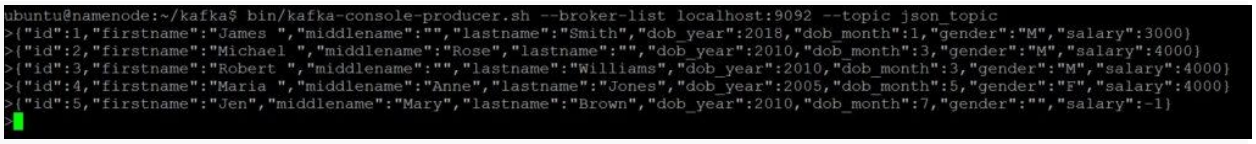
1. Run Kafka Producer Shell

First, let’s produce some JSON data to Kafka topic `"json_topic"`, Kafka distribution comes with Kafka Producer shell, run this producer and input the JSON data from person.json. Just copy one line at a time from person.json file and paste it on the console where Kafka Producer shell is running.

Note: By default when you write a message to a topic, Kafka automatically creates a topic however, you can also create a topic manually and specify your partition and replication factor.

```
bin/kafka-console-producer.sh \
--broker-list localhost:9092 --topic json_topic
```

Copy



```
ubuntu@namenode:~/kafka$ bin/kafka-console-producer.sh --broker-list localhost:9092 --topic json_topic
>{"id":1,"firstname":"James ","middlename":"","lastname":"Smith","dob_year":2018,"dob_month":1,"gender":"M","salary":3000}
>{"id":2,"firstname":"Michael ","middlename":"Rose","lastname":"","dob_year":2010,"dob_month":3,"gender":"M","salary":4000}
>{"id":3,"firstname":"Robert ","middlename":"","lastname":"Williams","dob_year":2010,"dob_month":3,"gender":"M","salary":4000}
>{"id":4,"firstname":"Maria ","middlename":"Anne","lastname":"Jones","dob_year":2005,"dob_month":5,"gender":"F","salary":4000}
>{"id":5,"firstname":"Jen","middlename":"Mary","lastname":"Brown","dob_year":2010,"dob_month":7,"gender":"","salary":-1}
>
```

2. Streaming With Kafka

2.1. Kafka Maven dependency

In order to streaming data from Kafka topic, we need to use below Kafka client Maven dependencies. You use the version according to yo your Kafka and Scala versions

```
<dependency>
     <groupId>org.apache.spark</groupId>
     <artifactId>spark-sql-kafka-0-10_2.11</artifactId>
     <version>2.4.0</version>
</dependency>
```

2.2 Spark Streaming Scala example

Spark Streaming uses `readStream()` on SparkSession to load a streaming Dataset from Kafka. Option `startingOffsets earliest` is used to read all data available in the Kafka at the start of the query, we may not use this option that often and the default value for `startingOffsets` is `latest` which reads only new data that's not been processed.

```
val df = spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "192.168.1.100:9092")
        .option("subscribe", "json_topic")
        .option("startingOffsets", "earliest") // From starting
        .load()

df.printSchema()
```

Copy

Since there are multiple options to stream from, we need to explicitly state from where you are streaming with `format("kafka")` and should provide the Kafka servers and subscribe to the topic you are streaming from using the option.

`df.printSchema()` returns the schema of streaming data from Kafka. The returned DataFrame contains all the familiar fields of a Kafka record and its associated metadata.

```
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

3. Spark Streaming Write to Console

Since the value is in binary, first we need to convert the binary value to String using `selectExpr()`

```
val personStringDF = df.selectExpr("CAST(value AS STRING)")
```

Copy

Now, extract the value which is in JSON String to DataFrame and convert to DataFrame columns using custom schema.

```
val schema = new StructType()
      .add("id",IntegerType)
      .add("firstname",StringType)
      .add("middlename",StringType)
      .add("lastname",StringType)
      .add("dob_year",IntegerType)
      .add("dob_month",IntegerType)
      .add("gender",StringType)
      .add("salary",IntegerType)

 val personDF = personStringDF.select(from_json(col("value"), schema).as("data"))
   .select("data.*")
```

Copy

```
personDF.writeStream
      .format("console")
      .outputMode("append")
      .start()
      .awaitTermination()
```

Copy

The complete Streaming Kafka Example code can be downloaded from GitHub. After download, import project to your favorite IDE and change Kafka broker IP address to your server IP on `SparkStreamingConsumerKafkaJson.scala` program. When you run this program, you should see Batch: 0 with data. As you input new data(from step 1), results get updated with Batch: 1, Batch: 2 and so on.

Batch: 0

```
+---+---------+----------+--------+----+------+------+
| id|firstname|middlename|lastname| dob|gender|salary|
+---+---------+----------+--------+----+------+------+
|  1|   James |          |   Smith|null|     M|  3000|
|  2| Michael |      Rose|        |null|     M|  4000|
|  3|  Robert |          |Williams|null|     M|  4000|
|  4|   Maria |      Anne|   Jones|null|     F|  4000|
+---+---------+----------+--------+----+------+------+

Batch: 1
+---+---------+----------+--------+---+------+------+
| id|firstname|middlename|lastname|dob|gender|salary|
+---+---------+----------+--------+---+------+------+
+---+---------+----------+--------+---+------+------+

Batch: 2
+---+---------+----------+--------+----+------+------+
| id|firstname|middlename|lastname| dob|gender|salary|
+---+---------+----------+--------+----+------+------+
|  1|   James |          |   Smith|null|     M|  3000|
+---+---------+----------+--------+----+------+------+
```

Copy

4. Spark Streaming Write to Kafka Topic

Note that In order to write Spark Streaming data to Kafka, `value` column is required and all other fields are optional.

columns `key` and `value` are binary in Kafka; hence, first, these should convert to String before processing. If a key column is not specified, then a `null` valued key column will be automatically added.

Let's produce the data to Kafka topic `"json_data_topic"`. Since we are processing JSON, let's convert data to JSON using `to_json()` function and store it in a value column.

```
df.selectExpr("CAST(id AS STRING) AS key", "to_json(struct(*)) AS value")
   .writeStream
   .format("kafka")
   .outputMode("append")
   .option("kafka.bootstrap.servers", "192.168.1.100:9092")
   .option("topic", "josn_data_topic")
   .start()
   .awaitTermination()
```

Copy

use `writeStream.format("kafka")` to write the streaming DataFrame to Kafka topic. Since we are just reading a file (without any aggregations) and writing as-is, we are using `outputMode("append")`. OutputMode is used to what data will be written to a sink when there is new data available in a DataFrame/Dataset

5. Run Kafka Consumer Shell

Now run the Kafka consumer shell program that comes with Kafka distribution.

```
bin/kafka-console-consumer.sh \
--broker-list localhost:9092 --topic josn_data_topic
```

Copy

As you feed more data (from step 1), you should see JSON output on the consumer shell console.

Spark Streaming – Different Output modes explained

- Post author:
- NNK
- Post category:
- Apache Spark / Apache Spark Streaming

Table of Contents

Let's see differences between complete, append and update output modes (`outputmode`) in Spark Streaming. `outputMode()` describes what data is written to a data sink (console, Kafka e.t.c) when there is new data available in streaming input (Kafka, Socket, e.t.c)

- Append Mode
- Complete Mode
- Update Mode

Streaming – Append Output Mode

OutputMode in which only the new rows in the streaming DataFrame/Dataset will be written to the sink.

This is the default mode. Use `append` as output mode `outputMode("append")` when you want to output only new rows to the output sink.

```
dF.writeStream
```

```
    .format("console")
    .outputMode("append")
    .start()
    .awaitTermination()
```

Copy

Streaming – Complete Output Mode

OutputMode in which all the rows in the streaming DataFrame/Dataset will be written to the sink every time there are some updates.

Use complete as output mode `outputMode("complete")` when you want to aggregate the data and output the entire results to sink every time. This mode is used only when you have streaming aggregated data. One example would be counting the words on streaming data and aggregating with previous data and output the results to sink.

```
val wordCountDF = df.select(explode(split(col("value")," ")).alias("word"))
    .groupBy("word").count()

wordCountDF.writeStream
        .format("console")
        .outputMode("complete")
        .start()
        .awaitTermination()
```

Copy

In case, if you use complete mode on non-aggregated stream, below exception would be thrown *“org.apache.spark.sql.AnalysisException: Complete output mode not supported when there are no streaming aggregations on streaming DataFrames/Datasets”*

Streaming – Update Output Mode

OutputMode in which only the rows that were updated in the streaming DataFrame/Dataset will be written to the sink every time there are some updates.

It is similar to the complete with one exception; update output mode `outputMode("update")` just outputs the updated aggregated results every time to data sink when new data arrives. but not the entire aggregated results like complete mode. If the streaming data is not aggregated then, it will act as append mode.

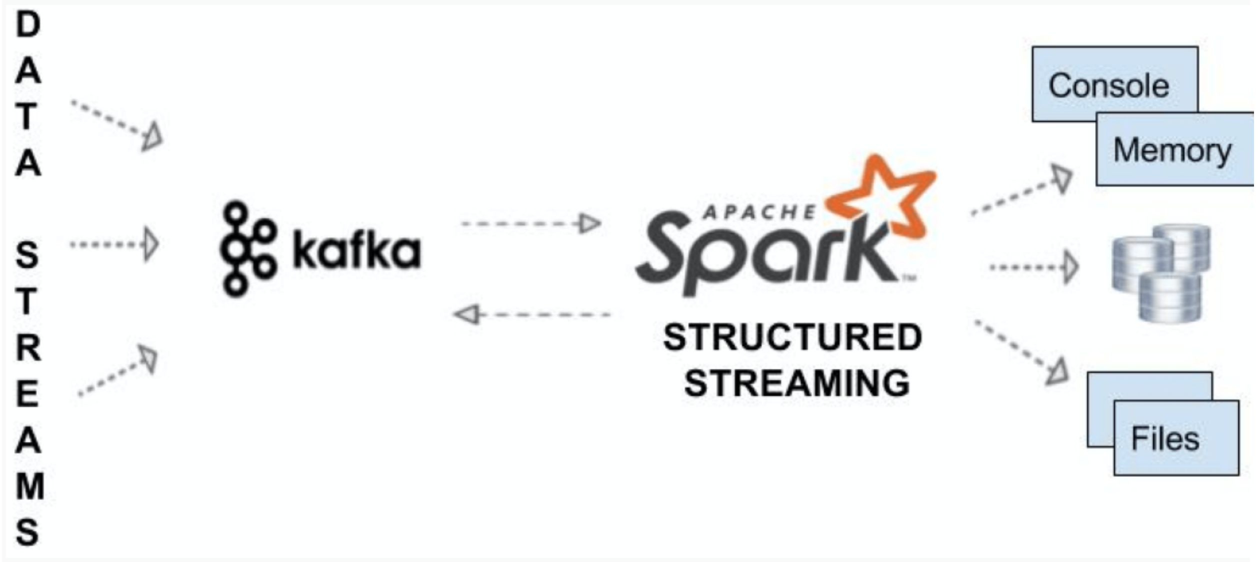
```
val wordCountDF = df.select(explode(split(col("value")," ")).alias("word"))
    .groupBy("word").count()

wordCountDF.writeStream
        .format("console")
        .outputMode("update")
        .start()
        .awaitTermination()
```

Spark Streaming – Kafka messages in Avro format

- Post author:
- NNK
- Post category:
- Apache Kafka / Apache Spark / Apache Spark Streaming

Table of Contents

This article describes Spark Structured Streaming from Kafka in Avro file format and usage of `from_avro()` and `to_avro()` SQL functions using the Scala programming language.



Spark Streaming Kafka messages in Avro

- Reading Avro Data from Kafka Topic
- Writing Avro Data to Kafka Topic
- How to Run
    - Running Producer with Example
    - Running Consumer with Example

Before deep-diving into this further let's understand a few points regarding Spark Streaming, Kafka and Avro.

Spark Streaming is a scalable, high-throughput, fault-tolerant streaming processing system that supports both batch and streaming workloads. It is an extension of the core Spark API to process real-time data from sources like Kafka, Flume, and Amazon Kinesis to name it a few. This processed data can be pushed to databases, Kafka, live dashboards e.t.c

Apache Kafka is a publish-subscribe messaging system originally written at LinkedIn. Accessing Kafka is enabled by using below Kafka client Maven dependency.

```
<dependency>
     <groupId>org.apache.spark</groupId>
     <artifactId>spark-sql-kafka-0-10_2.11</artifactId>
     <version>2.4.0</version>
</dependency>
```

Apache Avro is a data serialization system, it is mostly used in Apache Spark especially for Kafka-based data pipelines. When Avro data is stored in a file, its schema is stored with it, so that files may be processed later by any program.

Accessing Avro from Spark is enabled by using below Spark-Avro Maven dependency. The `spark-avro` module is external and not included in `spark-submit` or `spark-shell` by default.

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-avro_2.11</artifactId>
    <version>2.4.0</version>
</dependency>
```

Prerequisites

If you don't have Kafka cluster setup, follow the below articles to set up the single broker cluster and get familiar with creating and describing topics.

- Install & set-up Kafka Cluster guide
- How to create and describe Kafka topics

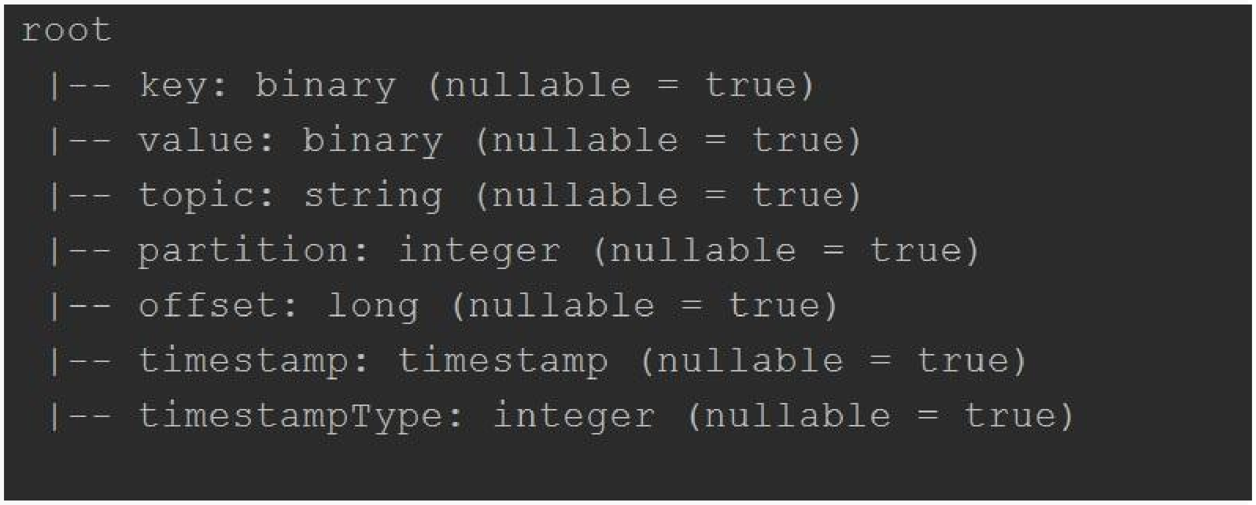
Reading Avro data from Kafka Topic

Streaming uses `readStream()` on SparkSession to load a streaming Dataset. `option("startingOffsets","earliest")` is used to read all data available in the topic at the start/earliest of the query, we may not use this option that often and the default value for `startingOffsets` is `latest` which reads only new data that's yet to process.

```
val df = spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "192.168.1.100:9092")
        .option("subscribe", "avro_topic")
        .option("startingOffsets", "earliest") // From starting
        .load()

df.printSchema()
```

Copy

`df.printSchema()` returns the schema of Kafka streaming. The returned DataFrame contains all the familiar fields of a Kafka record and its associated metadata.



```
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

When we are writing to Kafka, Value is required and all other fields are optional.

key and value are binary in Kafka; first, these should convert to String before we process. If a key column is not specified then a `null` valued key column will be automatically added.

To decode Avro data, we should use `from_avro()` function and this function takes Avro schema string as a parameter. For our example, I am going to load this schema from a person.avsc file. For reference, below is Avro's schema we going to use.

```
{
  "type": "record",
  "name": "Person",
  "namespace": "com.sparkbyexamples",
  "fields": [
    {"name": "id","type": ["int", "null"]},
    {"name": "firstname","type": ["string", "null"]},
    {"name": "middlename","type": ["string", "null"]},
    {"name": "lastname","type": ["string", "null"]},
    {"name": "dob_year","type": ["int", "null"]},
    {"name": "dob_month","type": ["int", "null"]},
    {"name": "gender","type": ["string", "null"]},
    {"name": "salary","type": ["int", "null"]}
  ]
}
```

Copy

The Schema defines the field names and data types. The receiver of Avro data needs to know this Schema one time before starting processing.

```
val jsonFormatSchema = new String(
Files.readAllBytes(Paths.get("./src/main/resources/person.avsc")))

val personDF = df.select(from_avro(col("value"),
jsonFormatSchema).as("person"))
        .select("person.*")
```

Copy

from_avro also takes a parameter that needs to decode. here we are decoding the Kafka value field.

Writing Avro data to Kafka Topic

Let's produce the data to Kafka topic `"avro_data_topic"`. Since we are processing Avro, let's encode data using `to_avro()` function and store it in a "value" column as Kafka needs data to be present in this field/column.

```
personDF.select(to_avro(struct("value")) as "value")
      .writeStream
      .format("kafka")
      .outputMode("append")
      .option("kafka.bootstrap.servers", "192.168.1.100:9092")
      .option("topic", "avro_data_topic")
      .option("checkpointLocation","c:/tmp")
      .start()
      .awaitTermination()
```

Copy

using `writeStream.format("kafka")` to write the streaming DataFrame to Kafka topic. Since we are just reading a file (without any aggregations) and writing as-is, we are using `outputMode("append")`. OutputMode is used to what data will be written to a sink when there is new data available in a DataFrame/Dataset

How to Run?

First will start a Kafka shell producer that comes with Kafka distribution and produces JSON message. later, I will write a Spark Streaming program that consumes these messages, converts it to Avro and sends it to another Kafka topic. Finally will create another Spark Streaming program that consumes Avro messages from Kafka, decodes the data to and writes it to Console.

1. Run Kafka Producer Shell

Run the Kafka Producer shell that comes with Kafka distribution and inputs the JSON data from person.json. To feed data, just copy one line at a time from person.json file and paste it on the console where Kafka Producer shell is running.

```
bin/kafka-console-producer.sh \
--broker-list localhost:9092 --topic json_topic
```

Copy

2. Run Kafka Producer

The complete Spark Streaming Avro Kafka Example code can be downloaded from GitHub. On this program change Kafka broker IP address to your server IP and run KafkaProduceAvro.scala from your favorite editor. This program reads the JSON message from Kafka topic `"json_topic"`, encode the data to Avro and sends it to another Kafka topic `"avro_topic"`.

```
package com.sparkbyexamples.spark.streaming.kafka.avro
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.{col, from_json,to_json,struct}
import org.apache.spark.sql.avro.to_avro
import org.apache.spark.sql.types.{IntegerType, StringType, StructType}
object KafkaProduceAvro {
  def main(args: Array[String]): Unit = {

    val spark: SparkSession = SparkSession.builder()
      .master("local[1]")
      .appName("SparkByExample.com")
      .getOrCreate()
    /*
    Disable logging as it writes too much log
     */
    spark.sparkContext.setLogLevel("ERROR")
    /*
    This consumes JSON data from Kafka
```

```
     */
    val df = spark.readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", "192.168.1.100:9092")
      .option("subscribe", "json_topic")
      .option("startingOffsets", "earliest") // From starting
      .load()
    /*
     Prints Kafka schema with columns (topic, offset, partition e.t.c)
      */
    df.printSchema()
    val schema = new StructType()
      .add("id",IntegerType)
      .add("firstname",StringType)
      .add("middlename",StringType)
      .add("lastname",StringType)
      .add("dob_year",IntegerType)
      .add("dob_month",IntegerType)
      .add("gender",StringType)
      .add("salary",IntegerType)
    /*
    Converts JSON string to DataFrame
     */
    val personDF = df.selectExpr("CAST(value AS STRING)") // First convert binary to string
      .select(from_json(col("value"), schema).as("data"))
    personDF.printSchema()

    /*
      * Convert DataFrame columns to Avro format and name it as "value"
      * And send this Avro data to Kafka topic
      */
    personDF.select(to_avro(struct("data.*")) as "value")
      .writeStream
      .format("kafka")
      .outputMode("append")
      .option("kafka.bootstrap.servers", "192.168.1.100:9092")
      .option("topic", "avro_topic")
      .option("checkpointLocation","c:/tmp")
      .start()
      .awaitTermination()
  }
}
```

Copy

3. Run Kafka Consumer

This program reads Avro message from Kafka topic `"avro_topics"`, decodes it and finally streams to console.

```
package com.sparkbyexamples.spark.streaming.kafka.avro
import java.nio.file.{Files, Paths}
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.avro._
import org.apache.spark.sql.functions.col

object KafkaConsumerAvro {
  def main(args: Array[String]): Unit = {

    val spark: SparkSession = SparkSession.builder()
      .master("local")
      .appName("SparkByExample.com")
      .getOrCreate()
    spark.sparkContext.setLogLevel("ERROR")
    val df = spark.readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", "192.168.1.100:9092")
      .option("subscribe", "avro_topic")
      .option("startingOffsets", "earliest") // From starting
      .load()
    /*
     Prints Kafka schema with columns (topic, offset, partition e.t.c)
      */
    df.printSchema()
    /*
    Read schema to convert Avro data to DataFrame
     */
    val jsonFormatSchema = new String(
      Files.readAllBytes(Paths.get("./src/main/resources/person.avsc")))

    val personDF = df.select(from_avro(col("value"), jsonFormatSchema).as("person"))
      .select("person.*")
    personDF.printSchema()
    /*
    Stream data to Console for testing
     */
    personDF.writeStream
      .format("console")
      .outputMode("append")
      .start()
      .awaitTermination()
  }
}
```

Spark from_avro() and to_avro() usage

- Post author:
- NNK
- Post category:
- Apache Spark

In Spark, `avro-module` is an external module and needed to add this module when processing Avro file and this `avro-module` provides function `to_avro()` to encode DataFrame column value to Avro binary format, and `from_avro()` to decode Avro binary data into a string value.

In this article, you will learn how to use from_avro() and to_avro() with Spark examples. mostly these functions are used in conjunction with Kafka when we need to read/write Kafka messages in Avro format hence, I will explain with Kafka context.

Let's assume that we have Kafka topic `avro_data_topic` and data to this topic (in Value field) are being sent in Avro format.

from_avro() – Reading Avro data from Kafka Topic

The `from_avro()` function from Spark module `spark-avro` is used to convert Avro binary format to string format.

Syntax

```
from_avro(data : org.apache.spark.sql.Column, jsonFormatSchema : scala.Predef.String) :
org.apache.spark.sql.Column
```

Copy

Spark uses `readStream()` on SparkSession to load a streaming Dataset from kafka topic. `option("startingOffsets","earliest")` is used to read all data available in the topic at the start/earliest of the query, we may not use this option that often and the default value for `startingOffsets` is `latest` which reads only new data that's yet to process.

```
val df = spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "192.168.1.100:9092")
        .option("subscribe", "avro_topic")
        .option("startingOffsets", "earliest") // From starting
        .load()
```

Copy

To decode Avro data, we should use `from_avro()` function and this function takes Avro schema string as a parameter. For our example, I am going to load this schema from a person.avsc file. For reference, below is Avro's schema we going to use.

```
{
  "type": "record",
  "name": "Person",
  "namespace": "com.sparkbyexamples",
  "fields": [
    {"name": "id","type": ["int", "null"]},
    {"name": "firstname","type": ["string", "null"]},
    {"name": "middlename","type": ["string", "null"]},
    {"name": "lastname","type": ["string", "null"]},
    {"name": "dob_year","type": ["int", "null"]},
    {"name": "dob_month","type": ["int", "null"]},
    {"name": "gender","type": ["string", "null"]},
    {"name": "salary","type": ["int", "null"]}
  ]
}
```

Copy

The Schema defines the field names and data types. The receiver of Avro data needs to know this Schema one time before starting processing.

```
val jsonFormatSchema = new String(
Files.readAllBytes(Paths.get("/src/main/resources/person.avsc")))

val personDF = df.select(from_avro(col("value"),
jsonFormatSchema).as("person"))
        .select("person.*")
```

Copy

to_avro() – Writing Avro data to Kafka Topic

Avro `to_avro()` function from the spark-avro module is used to convert a String value into Avro binary format.

Syntax

```
to_avro(data : org.apache.spark.sql.Column) : org.apache.spark.sql.Column
```

Copy

Let’s produce the data to Kafka topic `"avro_data_topic2"`. Since we are processing Avro message in Spark, we need to encode data using `to_avro()` function and store it in a “value” column as Kafka needs data to be present in this field/column.

```
personDF.select(to_avro(struct("value")) as "value")
    .writeStream
    .format("kafka")
    .outputMode("append")
    .option("kafka.bootstrap.servers", "192.168.1.100:9092")
    .option("topic", "avro_data_topic")
    .option("checkpointLocation","c:/tmp")
    .start()
    .awaitTermination()
```

Copy

using `writeStream.format("kafka")` to write the streaming DataFrame to Kafka topic. Since we are just reading a file (without any aggregations) and writing as-is, we are using `outputMode("append")`. OutputMode is used to what data will be written to a sink when there is new data available in a DataFrame/Dataset.

If you want a complete example and would like to execute and see the output, please access How to Stream Kafka messages in Avro format