|  |
| --- |
| Close-up image showing the leaf-sides of two oversized books side-by-side on a bookshelf, with additional books in soft focus background |
| PySpark with Examples |
| |  |  |  | | --- | --- | --- | |  | 6-Sep 2021 |  | |

Table of Contents

[PySpark Architecture 14](#_Toc81834669)

[Cluster Manager Types 14](#_Toc81834670)

[Spark Web UI – Understanding Spark Execution 15](#_Toc81834671)

[1. Spark Jobs Tab 17](#_Toc81834672)

[2. Stages Tab 18](#_Toc81834673)

[3. Tasks 22](#_Toc81834674)

[4. Storage 22](#_Toc81834675)

[5. Environment Tab 22](#_Toc81834676)

[6. Executors Tab 23](#_Toc81834677)

[7. SQL Tab 24](#_Toc81834678)

[RDD Tutorial 25](#_Toc81834679)

[What is RDD (Resilient Distributed Dataset)? 25](#_Toc81834680)

[RDD Benefits 25](#_Toc81834681)

[RDD Limitations 26](#_Toc81834682)

[Creating RDD 26](#_Toc81834683)

[RDD Parallelize 28](#_Toc81834684)

[Repartition and Coalesce 29](#_Toc81834685)

[RDD Operations 29](#_Toc81834686)

[Types of RDD 31](#_Toc81834687)

[Shuffle Operations 31](#_Toc81834688)

[RDD Persistence Tutorial 32](#_Toc81834689)

[Shared Variables Tutorial 34](#_Toc81834690)

[parallelize() – Create RDD from a list data 36](#_Toc81834691)

[Using sc.parallelize on PySpark Shell or REPL 36](#_Toc81834692)

[Using PySpark sparkContext.parallelize() in application 36](#_Toc81834693)

[create empty RDD by using sparkContext.parallelize 37](#_Toc81834694)

[Repartition() vs Coalesce() 37](#_Toc81834695)

[RDD Repartition() vs Coalesce() 37](#_Toc81834696)

[DataFrame repartition() vs coalesce() 39](#_Toc81834697)

[Broadcast Variables 41](#_Toc81834698)

[How does PySpark Broadcast work? 42](#_Toc81834699)

[How to create Broadcast variable 42](#_Toc81834700)

[PySpark RDD Broadcast variable example 42](#_Toc81834701)

[PySpark DataFrame Broadcast variable example 43](#_Toc81834702)

[PySpark Accumulator with Example 45](#_Toc81834703)

[Creating Accumulator Variable 45](#_Toc81834704)

[Accumulator Example 46](#_Toc81834705)

[Create DataFrame with Examples 47](#_Toc81834706)

[1. Create DataFrame from RDD 48](#_Toc81834707)

[2. Create DataFrame from List Collection 49](#_Toc81834708)

[3. Create DataFrame from Data sources 50](#_Toc81834709)

[4. Other sources (Avro, Parquet, ORC, Kafka) 51](#_Toc81834710)

[Create an Empty DataFrame & RDD 51](#_Toc81834711)

[1. Create Empty RDD in PySpark 52](#_Toc81834712)

[2. Create Empty DataFrame with Schema (StructType) 52](#_Toc81834713)

[3. Convert Empty RDD to DataFrame 53](#_Toc81834714)

[4. Create Empty DataFrame with Schema. 53](#_Toc81834715)

[5. Create Empty DataFrame without Schema (no columns) 53](#_Toc81834716)

[Convert PySpark RDD to DataFrame 54](#_Toc81834717)

[1. Create PySpark RDD 54](#_Toc81834718)

[2. Convert PySpark RDD to DataFrame 54](#_Toc81834719)

[2.3 Using createDataFrame() with StructType schema 55](#_Toc81834720)

[3. Complete Example 56](#_Toc81834721)

[4. Conclusion: 56](#_Toc81834722)

[Convert PySpark DataFrame to Pandas 57](#_Toc81834723)

[Prepare PySpark DataFrame 57](#_Toc81834724)

[Convert PySpark Dataframe to Pandas DataFrame 58](#_Toc81834725)

[Convert Spark Nested Struct DataFrame to Pandas 58](#_Toc81834726)

[PySpark show() – Display DataFrame Contents in Table 60](#_Toc81834727)

[1. PySpark DataFrame show() Syntax & Example 60](#_Toc81834728)

[1.2 Example 60](#_Toc81834729)

[PySpark StructType & StructField Explained with Examples 62](#_Toc81834730)

[1. StructType – Defines the structure of the Dataframe 62](#_Toc81834731)

[2. StructField – Defines the metadata of the DataFrame column 62](#_Toc81834732)

[3. Using PySpark StructType & StructField with DataFrame 62](#_Toc81834733)

[4. Defining Nested StructType object struct 64](#_Toc81834734)

[5. Adding & Changing struct of the DataFrame 65](#_Toc81834735)

[6. Using SQL ArrayType and MapType 65](#_Toc81834736)

[7. Creating StructType object struct from JSON file 66](#_Toc81834737)

[8. Creating StructType object struct from DDL String 67](#_Toc81834738)

[9. Checking if a Column Exists in a DataFrame 67](#_Toc81834739)

[10. Complete Example of PySpark StructType & StructField 68](#_Toc81834740)

[Row using on DataFrame and RDD 70](#_Toc81834741)

[1. Create a Row Object 70](#_Toc81834742)

[2. Create Custom Class from Row 70](#_Toc81834743)

[3. Using Row class on PySpark RDD 71](#_Toc81834744)

[4. Using Row class on PySpark DataFrame 71](#_Toc81834745)

[5. Create Nested Struct Using Row Class 72](#_Toc81834746)

[6. Complete Example of PySpark Row usage on RDD & DataFrame 73](#_Toc81834747)

[Column Class | Operators & Functions 74](#_Toc81834748)

[1. Create Column Class Object 75](#_Toc81834749)

[2. PySpark Column Operators 76](#_Toc81834750)

[3. PySpark Column Functions 76](#_Toc81834751)

[4. PySpark Column Functions Examples 78](#_Toc81834752)

[4.1 alias() – Set’s name to Column 78](#_Toc81834753)

[4.3 cast() & astype() – Used to convert the data Type. 79](#_Toc81834754)

[PySpark Select Columns From DataFrame 82](#_Toc81834755)

[1. Select Single & Multiple Columns From PySpark 82](#_Toc81834756)

[2. Select All Columns From List 82](#_Toc81834757)

[3. Select Columns by Index 83](#_Toc81834758)

[4. Select Nested Struct Columns from PySpark 83](#_Toc81834759)

[5. Complete Example 85](#_Toc81834760)

[6. Conclusion 86](#_Toc81834761)

[Related Articles 86](#_Toc81834762)

[PySpark Collect() – Retrieve data from DataFrame 87](#_Toc81834763)

[When to avoid Collect() 88](#_Toc81834764)

[collect () vs select () 88](#_Toc81834765)

[Complete Example of PySpark collect() 89](#_Toc81834766)

[Conclusion 89](#_Toc81834767)

[PySpark withColumn() Usage with Examples 90](#_Toc81834768)

[1. Change DataType using PySpark withColumn() 90](#_Toc81834769)

[2. Update The Value of an Existing Column 90](#_Toc81834770)

[3. Create a Column from an Existing 90](#_Toc81834771)

[4. Add a New Column using withColumn() 91](#_Toc81834772)

[5. Rename Column Name 91](#_Toc81834773)

[6. Drop Column From PySpark DataFrame 91](#_Toc81834774)

[7. PySpark withColumn() Complete Example 92](#_Toc81834775)

[PySpark withColumnRenamed to Rename Column on DataFrame 93](#_Toc81834776)

[1. PySpark withColumnRenamed – To rename DataFrame column name 94](#_Toc81834777)

[2. PySpark withColumnRenamed – To rename multiple columns 94](#_Toc81834778)

[3. Using PySpark StructType – To rename a nested column in Dataframe 95](#_Toc81834779)

[4. Using Select – To rename nested elements. 95](#_Toc81834780)

[5. Using PySpark DataFrame withColumn – To rename nested columns 96](#_Toc81834781)

[6. Using col() function – To Dynamically rename all or multiple columns 96](#_Toc81834782)

[7. Using toDF() – To change all columns in a PySpark DataFrame 96](#_Toc81834783)

[Source code 96](#_Toc81834784)

[PySpark Where Filter Function | Multiple Conditions 99](#_Toc81834785)

[1. PySpark DataFrame filter() Syntax 101](#_Toc81834786)

[2. DataFrame filter() with Column Condition 102](#_Toc81834787)

[3. DataFrame filter() with SQL Expression 103](#_Toc81834788)

[4. PySpark Filter with Multiple Conditions 103](#_Toc81834789)

[5. Filter Based on List Values 103](#_Toc81834790)

[6. Filter Based on Starts With, Ends With, Contains 104](#_Toc81834791)

[7. PySpark Filter like and rlike 104](#_Toc81834792)

[8. Filter on an Array column 105](#_Toc81834793)

[9. Filtering on Nested Struct columns 105](#_Toc81834794)

[10. Source code of PySpark where filter 106](#_Toc81834795)

[11. Conclusion 107](#_Toc81834796)

[PySpark – Distinct to Drop Duplicate Rows 107](#_Toc81834797)

[1. Get Distinct Rows (By Comparing All Columns) 108](#_Toc81834798)

[2. PySpark Distinct of Selected Multiple Columns 109](#_Toc81834799)

[3. Source Code to Get Distinct Rows 110](#_Toc81834800)

[Conclusion 110](#_Toc81834801)

[PySpark orderBy() and sort() explained 111](#_Toc81834802)

[DataFrame sorting using the sort() function 112](#_Toc81834803)

[DataFrame sorting using orderBy() function 112](#_Toc81834804)

[Sort by Ascending (ASC) 112](#_Toc81834805)

[Sort by Descending (DESC) 113](#_Toc81834806)

[Using Raw SQL 113](#_Toc81834807)

[Dataframe Sorting Complete Example 114](#_Toc81834808)

[Conclusion 114](#_Toc81834809)

[PySpark Groupby Explained with Example 115](#_Toc81834810)

[Preparing Data & creating DataFrame 115](#_Toc81834811)

[PySpark groupBy and aggregate on DataFrame columns 116](#_Toc81834812)

[PySpark groupBy and aggregate on multiple columns 116](#_Toc81834813)

[Running more aggregates at a time 117](#_Toc81834814)

[Using filter on aggregate data 118](#_Toc81834815)

[PySpark groupBy Example Source code 118](#_Toc81834816)

[Conclusion 119](#_Toc81834817)

[PySpark Join Types | Join Two DataFrames 119](#_Toc81834818)

[1. PySpark Join Syntax 119](#_Toc81834819)

[2. PySpark Join Types 120](#_Toc81834820)

[3. PySpark Inner Join DataFrame 121](#_Toc81834821)

[4. PySpark Full Outer Join 122](#_Toc81834822)

[5. PySpark Left Outer Join 123](#_Toc81834823)

[6. Right Outer Join 123](#_Toc81834824)

[7. Left Semi Join 124](#_Toc81834825)

[8. Left Anti Join 124](#_Toc81834826)

[9. PySpark Self Join 125](#_Toc81834827)

[4. Using SQL Expression 125](#_Toc81834828)

[5. PySpark SQL Join on multiple DataFrames 126](#_Toc81834829)

[6. PySpark SQL Join Complete Example 126](#_Toc81834830)

[PySpark Union and UnionAll Explained 128](#_Toc81834831)

[Merge two or more DataFrames using union 129](#_Toc81834832)

[Merge DataFrames using unionAll 130](#_Toc81834833)

[Merge without Duplicates 130](#_Toc81834834)

[Complete Example of DataFrame Union 131](#_Toc81834835)

[Conclusion 131](#_Toc81834836)

[Spark Merge Two DataFrames with Different Columns or Schema 132](#_Toc81834837)

[Spark Merge Two DataFrames with Different Columns 132](#_Toc81834838)

[PySpark Merge Two DataFrames with Different Columns 133](#_Toc81834839)

[Conclusion 134](#_Toc81834840)

[PySpark UDF (User Defined Function) 135](#_Toc81834841)

[1. PySpark UDF Introduction 135](#_Toc81834842)

[2. Create PySpark UDF 136](#_Toc81834843)

[3. Using UDF with DataFrame 137](#_Toc81834844)

[4. Creating UDF using annotation 138](#_Toc81834845)

[5. Special Handling 138](#_Toc81834846)

[6. Complete PySpark UDF Example 140](#_Toc81834847)

[Conclusion 142](#_Toc81834848)

[PySpark map() Transformation 142](#_Toc81834849)

[map() Syntax 142](#_Toc81834850)

[PySpark map() Example with RDD 143](#_Toc81834851)

[PySpark map() Example with DataFrame 143](#_Toc81834852)

[Complete PySpark map() example 145](#_Toc81834853)

[PySpark flatMap() Transformation 146](#_Toc81834854)

[flatMap() Syntax 147](#_Toc81834855)

[flatMap() Example 147](#_Toc81834856)

[Complete PySpark flatMap() example 147](#_Toc81834857)

[Using flatMap() transformation on DataFrame 148](#_Toc81834858)

[PySpark – Loop/Iterate Through Rows in DataFrame 149](#_Toc81834859)

[PySpark Loop Through Rows in DataFrame Examples 149](#_Toc81834860)

[Using map() to Loop Through Rows in DataFrame 150](#_Toc81834861)

[Using foreach() to Loop Through Rows in DataFrame 151](#_Toc81834862)

[Using pandas() to Iterate 151](#_Toc81834863)

[Collect Data As List and Loop Through 151](#_Toc81834864)

[PySpark Random Sample with Example 152](#_Toc81834865)

[1. PySpark SQL sample() Usage & Examples 152](#_Toc81834866)

[2. PySpark RDD Sample 154](#_Toc81834867)

[PySpark fillna() & fill() – Replace NULL/None Values 155](#_Toc81834868)

[PySpark fillna() & fill() Syntax 156](#_Toc81834869)

[PySpark Replace NULL/None Values with Zero (0) 156](#_Toc81834870)

[PySpark Replace Null/None Value with Empty String 156](#_Toc81834871)

[Complete Code 158](#_Toc81834872)

[Conclusion 158](#_Toc81834873)

[Reference: 158](#_Toc81834874)

[PySpark Pivot and Unpivot DataFrame 159](#_Toc81834875)

[Pivot PySpark DataFrame 160](#_Toc81834876)

[Pivot Performance improvement in PySpark 2.0 160](#_Toc81834877)

[Unpivot PySpark DataFrame 161](#_Toc81834878)

[Transpose or Pivot without aggregation 161](#_Toc81834879)

[Complete Example 161](#_Toc81834880)

[PySpark partitionBy() – Write to Disk Example 162](#_Toc81834881)

[1. What is PySpark Partition? 162](#_Toc81834882)

[2. Partition Advantages 163](#_Toc81834883)

[3. Create DataFrame 163](#_Toc81834884)

[4. PySpark partitionBy() 163](#_Toc81834885)

[5. PySpark partitionBy() Multiple Columns 164](#_Toc81834886)

[6. Using repartition() and partitionBy() together 165](#_Toc81834887)

[7. Data Skew – Control Number of Records per Partition File 165](#_Toc81834888)

[8. Read a Specific Partition 166](#_Toc81834889)

[9. PySpark SQL – Read Partition Data 166](#_Toc81834890)

[10. How to Choose a Partition Column When Writing to File system? 167](#_Toc81834891)

[PySpark ArrayType Column With Examples 167](#_Toc81834892)

[What is PySpark ArrayType 167](#_Toc81834893)

[Create PySpark ArrayType 168](#_Toc81834894)

[Create PySpark ArrayType Column Using StructType 168](#_Toc81834895)

[PySpark ArrayType (Array) Functions 169](#_Toc81834896)

[explode() 169](#_Toc81834897)

[Split() 169](#_Toc81834898)

[array() 170](#_Toc81834899)

[array\_contains() 170](#_Toc81834900)

[Conclusion 170](#_Toc81834901)

[PySpark MapType (Dict) Usage with Examples 171](#_Toc81834902)

[1. Create PySpark MapType 171](#_Toc81834903)

[2. Create MapType From StructType 171](#_Toc81834904)

[3. Access PySpark MapType Elements 172](#_Toc81834905)

[4. Functions 173](#_Toc81834906)

[PySpark Aggregate Functions with Examples 174](#_Toc81834907)

[PySpark Aggregate Functions 174](#_Toc81834908)

[PySpark Aggregate Functions Examples 175](#_Toc81834909)

[approx\_count\_distinct Aggregate Function 175](#_Toc81834910)

[avg (average) Aggregate Function 176](#_Toc81834911)

[collect\_list Aggregate Function 176](#_Toc81834912)

[collect\_set Aggregate Function 176](#_Toc81834913)

[countDistinct Aggregate Function 176](#_Toc81834914)

[count function 176](#_Toc81834915)

[grouping function 177](#_Toc81834916)

[first function 177](#_Toc81834917)

[last function 177](#_Toc81834918)

[kurtosis function 177](#_Toc81834919)

[max function 178](#_Toc81834920)

[min function 178](#_Toc81834921)

[mean function 178](#_Toc81834922)

[skewness function 178](#_Toc81834923)

[stddev(), stddev\_samp() and stddev\_pop() 179](#_Toc81834924)

[sum function 179](#_Toc81834925)

[sumDistinct function 179](#_Toc81834926)

[variance(), var\_samp(), var\_pop() 179](#_Toc81834927)

[Source code of PySpark Aggregate examples 180](#_Toc81834928)

[Conclusion 181](#_Toc81834929)

[PySpark Window Functions 181](#_Toc81834930)

[1. Window Functions 181](#_Toc81834931)

[2. PySpark Window Ranking functions 184](#_Toc81834932)

[3. PySpark Window Analytic functions 187](#_Toc81834933)

[4. PySpark Window Aggregate Functions 188](#_Toc81834934)

[Source Code of Window Functions Example 189](#_Toc81834935)

[Conclusion 190](#_Toc81834936)

[PySpark SQL Date and Timestamp Functions 191](#_Toc81834937)

[PySpark SQL Date Functions 191](#_Toc81834938)

[PySpark SQL Timestamp Functions 193](#_Toc81834939)

[Date and Timestamp Window Functions 194](#_Toc81834940)

[PySpark SQL Date and Timestamp Functions Examples 195](#_Toc81834941)

[hour(), Minute() and second() 200](#_Toc81834942)

[PySpark JSON Functions with Examples 201](#_Toc81834943)

[1. PySpark JSON Functions 201](#_Toc81834944)

[2. PySpark JSON Functions Examples 201](#_Toc81834945)

[2.1. from\_json() 201](#_Toc81834946)

[2.2. to\_json() 202](#_Toc81834947)

[2.3. json\_tuple() 202](#_Toc81834948)

[2.4. get\_json\_object() 203](#_Toc81834949)

[2.5. schema\_of\_json() 203](#_Toc81834950)

[3. Complete Example of PySpark JSON Functions 203](#_Toc81834951)

[PySpark Read CSV file into DataFrame 204](#_Toc81834952)

[1. PySpark Read CSV File into DataFrame 204](#_Toc81834953)

[2. Options While Reading CSV File 206](#_Toc81834954)

[3. Reading CSV files with a user-specified custom schema 207](#_Toc81834955)

[4. Applying DataFrame transformations 207](#_Toc81834956)

[5. Write PySpark DataFrame to CSV file 207](#_Toc81834957)

[6. PySpark Read CSV Complete Example 208](#_Toc81834958)

[7. Conclusion: 209](#_Toc81834959)

[PySpark Read and Write Parquet File 210](#_Toc81834960)

[What is Parquet File? 210](#_Toc81834961)

[Apache Parquet Pyspark Example 210](#_Toc81834962)

[Pyspark Write DataFrame to Parquet file format 211](#_Toc81834963)

[Pyspark Read Parquet file into DataFrame 211](#_Toc81834964)

[Append or Overwrite an existing Parquet file 211](#_Toc81834965)

[Executing SQL queries DataFrame 211](#_Toc81834966)

[Creating a table on Parquet file 212](#_Toc81834967)

[Create Parquet partition file 212](#_Toc81834968)

[Retrieving from a partitioned Parquet file 213](#_Toc81834969)

[Creating a table on Partitioned Parquet file 213](#_Toc81834970)

[Complete Example of PySpark read and write Parquet file 213](#_Toc81834971)

[PySpark Read JSON file into DataFrame 215](#_Toc81834972)

[PySpark Read JSON file into DataFrame 215](#_Toc81834973)

[Read JSON file from multiline 215](#_Toc81834974)

[Reading multiple files at a time 216](#_Toc81834975)

[Reading all files in a directory 216](#_Toc81834976)

[Reading files with a user-specified custom schema 216](#_Toc81834977)

[Read JSON file using PySpark SQL 217](#_Toc81834978)

[Options while reading JSON file 217](#_Toc81834979)

[Applying DataFrame transformations 217](#_Toc81834980)

[Write PySpark DataFrame to JSON file 217](#_Toc81834981)

[Source code for reference 218](#_Toc81834982)

[Conclusion: 219](#_Toc81834983)

[Spark Read XML file 220](#_Toc81834984)

[Read & Write Avro files using Spark 224](#_Toc81834985)

[Hive Tables 225](#_Toc81834986)

[JDBC To Other Databases 229](#_Toc81834987)

[Spark Streaming with Kafka Example 233](#_Toc81834988)

[Regular expression 234](#_Toc81834989)

[PySpark When Otherwise | SQL Case When Usage 240](#_Toc81834990)

[2. PySpark SQL Case When on DataFrame. 241](#_Toc81834991)

[2.3. Multiple Conditions using & and | operator 243](#_Toc81834992)

[Complete Example – PySpark When Otherwise | SQL Case When 243](#_Toc81834993)

[PySpark SQL expr() (Expression ) Function 244](#_Toc81834994)

[1. PySpark expr() Syntax 245](#_Toc81834995)

[2. PySpark SQL expr() Function Examples 245](#_Toc81834996)

[3. Complete Example of PySpark SQL expr() Function 247](#_Toc81834997)

[Conclusion 248](#_Toc81834998)

[References 248](#_Toc81834999)

[PySpark lit() – Add Literal or Constant to DataFrame 249](#_Toc81835000)

[lit() Function to Add Constant Column 249](#_Toc81835001)

[typedLit() Function – Syntax 250](#_Toc81835002)

[Complete Example of How to Add Constant Column 250](#_Toc81835003)

[PySpark Convert String to Array Column 251](#_Toc81835004)

[Split() function syntax 251](#_Toc81835005)

[PySpark Convert String to Array Column 252](#_Toc81835006)

[Convert String to Array Column using SQL Query 252](#_Toc81835007)

[Complete Example 253](#_Toc81835008)

[PySpark – Convert array column to a String 253](#_Toc81835009)

[Convert an array of String to String column using concat\_ws() 254](#_Toc81835010)

[Using PySpark SQL expression 255](#_Toc81835011)

[Complete Example 255](#_Toc81835012)

[Pyspark – Get substring() from a column 256](#_Toc81835013)

[Using SQL function substring() 256](#_Toc81835014)

[2. Using substring() with select() 256](#_Toc81835015)

[3.Using substring() with selectExpr() 257](#_Toc81835016)

[4. Using substr() from Column type 257](#_Toc81835017)

[Complete Example of PySpark substring() 257](#_Toc81835018)

[PySpark Replace Column Values in DataFrame 258](#_Toc81835019)

[1. PySpark Replace String Column Values 258](#_Toc81835020)

[2. Replace Column Values Conditionally 259](#_Toc81835021)

[3. Replace Column Value with Dictionary (map) 259](#_Toc81835022)

[4. Replace Column Value Character by Character 259](#_Toc81835023)

[5. Replace Column with Another Column Value 260](#_Toc81835024)

[6. Replace All or Multiple Column Values 260](#_Toc81835025)

[7. Using overlay() Function 260](#_Toc81835026)

[8. Complete Example 261](#_Toc81835027)

[References 262](#_Toc81835028)

[PySpark to\_timestamp() – Convert String to Timestamp type 262](#_Toc81835029)

[Convert String to PySpark Timestamp type 262](#_Toc81835030)

[Custom string format to Timestamp type 263](#_Toc81835031)

[SQL Example 263](#_Toc81835032)

[Complete Example for quick reference 264](#_Toc81835033)

[PySpark to\_date() – Convert Timestamp to Date 265](#_Toc81835034)

[Using to\_date() – Convert Timestamp String to Date 265](#_Toc81835035)

[Convert TimestampType (timestamp) to DateType (date) 266](#_Toc81835036)

[Using Column cast() Function 266](#_Toc81835037)

[PySpark SQL – Convert Timestamp to Date 267](#_Toc81835038)

[Complete code 267](#_Toc81835039)

[PySpark date\_format() – Convert Date to String format 268](#_Toc81835040)

[Complete Example of Convert Date to String 269](#_Toc81835041)

[PySpark – Difference between two dates (days, months, years) 270](#_Toc81835042)

[datediff() Function 270](#_Toc81835043)

[months\_between() Function 271](#_Toc81835044)

[SQL Example 272](#_Toc81835045)

[PySpark Convert DataFrame Columns to MapType (Dict) 273](#_Toc81835046)

[Convert DataFrame Columns to MapType 274](#_Toc81835047)

[PySpark Count Distinct from DataFrame 275](#_Toc81835048)

[Using DataFrame distinct() and count() 276](#_Toc81835049)

[Using countDistinct() SQL Function 276](#_Toc81835050)

[Using SQL to get Count Distinct 277](#_Toc81835051)

[Source Code of PySpark Count Distinct Example 277](#_Toc81835052)

[Conclusion 277](#_Toc81835053)

[PySpark lit() – Add Literal or Constant to DataFrame 278](#_Toc81835054)

[lit() Function to Add Constant Column 278](#_Toc81835055)

[typedLit() Function – Syntax 279](#_Toc81835056)

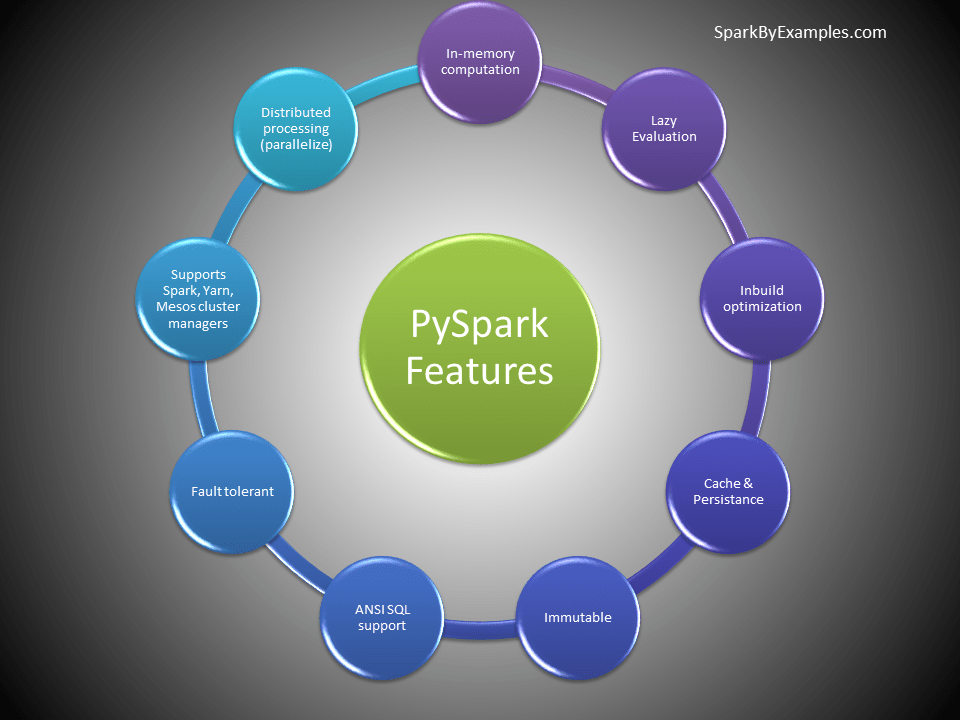
[Complete Example of How to Add Constant Column 279](#_Toc81835057)

### Who uses PySpark?

PySpark is very well used in Data Science and Machine Learning community as there are many widely used data science libraries written in Python including NumPy, TensorFlow also used due to its efficient processing of large datasets. PySpark has been used by many organizations like Walmart, Trivago, Sanofi, Runtastic, and many more.

### Features

Following are the main features of PySpark.

PySpark Features

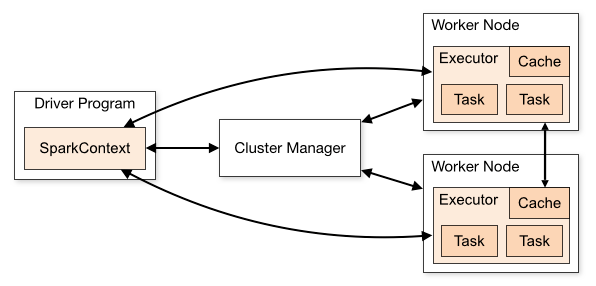
* In-memory computation
* Distributed processing using parallelize
* Can be used with many cluster managers (Spark, Yarn, Mesos e.t.c)
* Fault-tolerant
* Immutable
* Lazy evaluation
* Cache & persistence
* Inbuild-optimization when using DataFrames
* Supports ANSI SQL

### Advantages of PySpark

* PySpark is a general-purpose, in-memory, distributed processing engine that allows you to process data efficiently in a distributed fashion.
* Applications running on PySpark are 100x faster than traditional systems.
* You will get great benefits using PySpark for data ingestion pipelines.
* Using PySpark we can process data from Hadoop HDFS, AWS S3, and many file systems.
* PySpark also is used to process real-time data using Streaming and Kafka.
* Using PySpark streaming you can also stream files from the file system and also stream from the socket.
* PySpark natively has machine learning and graph libraries.

## 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](https://spark.apache.org/docs/latest/spark-standalone.html) – a simple cluster manager included with Spark that makes it easy to set up a cluster.
* [Apache Mesos](https://spark.apache.org/docs/latest/running-on-mesos.html) – Mesons is a Cluster manager that can also run Hadoop MapReduce and PySpark applications.
* [Hadoop YARN](https://spark.apache.org/docs/latest/running-on-yarn.html) – the resource manager in Hadoop 2. This is mostly used, cluster manager.
* [Kubernetes](https://spark.apache.org/docs/latest/running-on-kubernetes.html) – 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.

**Spark Web UI – Understanding Spark Execution**

Apache Spark provides a suite of Web UI/User Interfaces ([Jobs](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#spark-jobs), [Stages](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#spark-stages), [Tasks](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#tasks), [Storage](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#storage), [Environment](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#environment), [Executors](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#executors), and [SQL](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#sql)) to monitor the status of your Spark/PySpark application, resource consumption of Spark cluster, and Spark configurations.

To better understand how Spark executes the Spark/PySpark Jobs, these set of user interfaces comes in handy. In this article, I will run a small application and explain how Spark executes this by using different sections in Spark Web UI.

Before going into Spark UI first, learn about these two concepts.

* [Transformations](https://sparkbyexamples.com/apache-spark-rdd/spark-rdd-transformations/)
* [Action](https://sparkbyexamples.com/apache-spark-rdd/spark-rdd-actions/)

Let me give a small brief on those two, Your application code is the set of instructions that instructs the driver to do a Spark Job and let the driver decide how to achieve it with the help of executors.

Instructions to the driver are called Transformations and action will trigger the execution.

I had written a small application which does transformation and action.

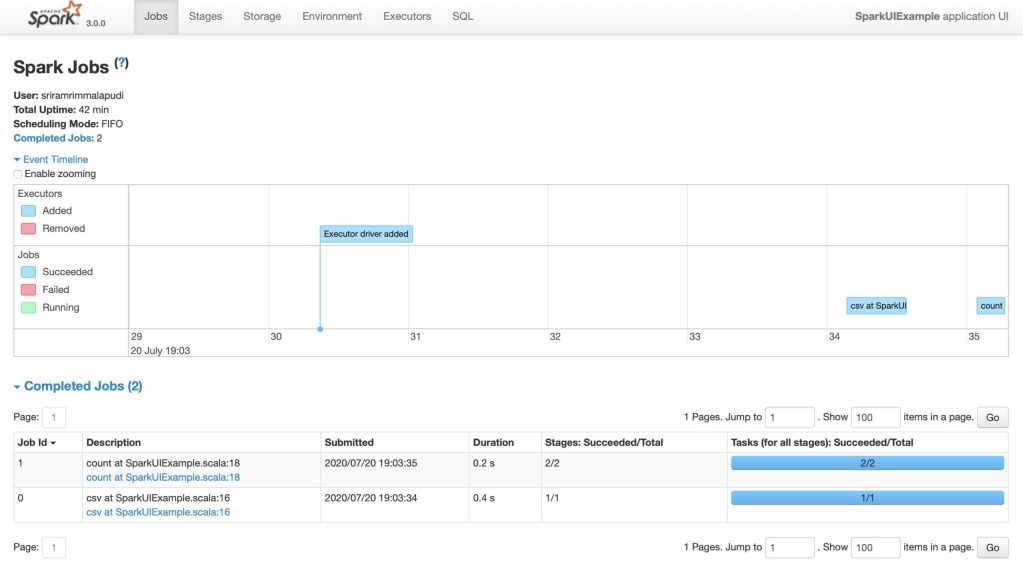
Application Code

Here we are [creating a DataFrame](https://sparkbyexamples.com/spark/different-ways-to-create-a-spark-dataframe/) by [reading a .csv file](https://sparkbyexamples.com/apache-spark-rdd/spark-load-csv-file-into-rdd/) and checking the count of the DataFrame. Let’s understand how an application gets projected in Spark UI

Spark UI is separated into below tabs.

1. [Spark Jobs](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#spark-jobs)
2. [Stages](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#spark-stages)
3. [Tasks](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#tasks)
4. [Storage](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#storage)
5. [Environment](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#environment)
6. [Executors](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#executors)
7. [SQL](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#sql)

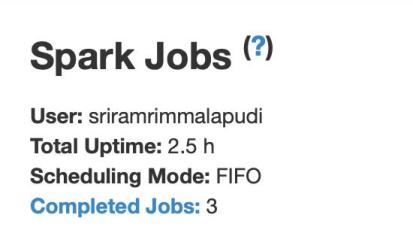
If you are running the Spark application locally, Spark UI can be accessed using the <http://localhost:4040/> . Spark UI by default runs on port 4040 and below are some of the additional UI’s that would be helpful to track Spark application.

Spark Web UI

* Spark Application UI: <http://localhost:4040/>
* Resource Manager: [http://localhost:9870](http://localhost:9870/)
* Spark JobTracker: <http://localhost:8088/>
* Node Specific Info: <http://localhost:8042/>

**Note:** To access these URLs, Spark application should in running state. If you wanted to access this URL regardless of your Spark application status and wanted to access Spark UI all the time, you would need to start [Spark History server](https://sparkbyexamples.com/hadoop/spark-setup-on-yarn/#spark-history-server).

**1. Spark Jobs Tab**

Jobs tab

The details that I want you to be aware of under the jobs section are [**Scheduling** **mode**](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#scheduling), the [**number of Spark Jobs**](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#Job), the [**number of stages**](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#Stage) it has, and [**Description**](https://sparkbyexamples.com/spark/spark-web-ui-understanding/#Description) in your spark job.

**1.1 Scheduling Mode**

We have three Scheduling modes.

1. **Standalone**mode
2. **YARN** mode
3. **Mesos**

Spark Scheduling tab

As I was running in a local machine, I tried using Standalone mode

**1.2 Number of Spark Jobs:**

Always keep in mind, the number of Spark jobs is equal to the number of actions in the application and each Spark job should have at least one Stage.  
In our above application, we have performed 3 Spark jobs (0,1,2)

* Job *0. read the CSV file.*
* Job *1. Inferschema from the file.*
* Job *2. Count Check*

So if we look at the fig it clearly shows 3 Spark jobs result of 3 actions.

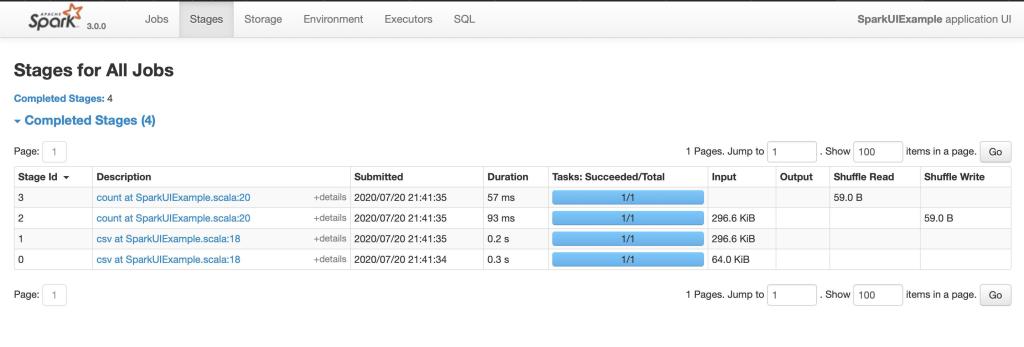
**1.3 Number of Stages**

Each [Wide Transformation](https://sparkbyexamples.com/apache-spark-rdd/spark-rdd-transformations/#wider-transformation) results in a separate Numberof Stages. In our case, Spark job0 and Spark job1 have individual single stages but when it comes to Spark job 3 we can see two stages that are because of the partition of data. Data is partitioned into two files by default.

**1.4 Description**

Description links the complete details of the associated SparkJob like Spark Job Status, DAG Visualization, Completed Stages  
I had explained the description part in the coming part.

**2. Stages Tab**

Spark Stage Tab

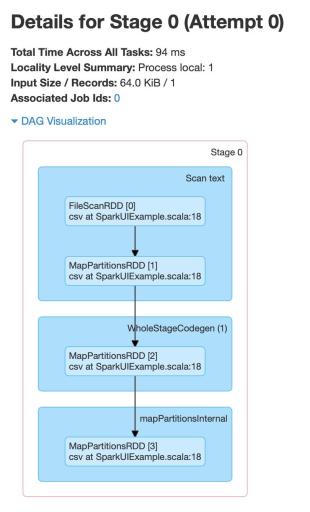
We can navigate into Stage Tab in two ways.

1. Select the Description of the respective Spark job (Shows stages only for the Spark job opted)
2. On the top of Spark Job tab select Stages option (Shows all stages in Application)

In our application, we have a total of **4 *Stages***.

The Stage tab displays a summary page that shows the current state of all stages of all Spark jobs in the spark application

The number of tasks you could see in each stage is the number of partitions that spark is going to work on and each task inside a stage is the same work that will be done by spark but on a different partition of data.

Stage 0

**Stage detail**

Details of stage showcase Directed Acyclic Graph (DAG) of this stage, where vertices represent the RDDs or DataFrame and edges represent an operation to be applied.

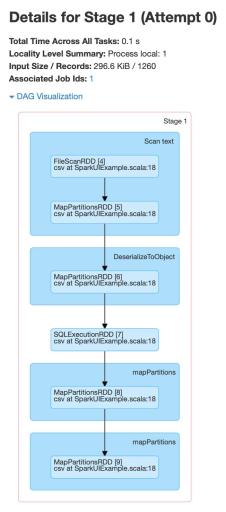
let us analyze operations in Stages  
Operations in Stage0 are  
1.FileScanRDD  
2.MapPartitionsRDD

**FileScanRDD**

FileScan represents reading the data from a file.  
It is  given FilePartitions that are custom RDD partitions with PartitionedFiles (file blocks)  
In our scenario, the *CSV file is read*

**MapPartitionsRDD**

MapPartitionsRDD will be created when you use map Partition transformation

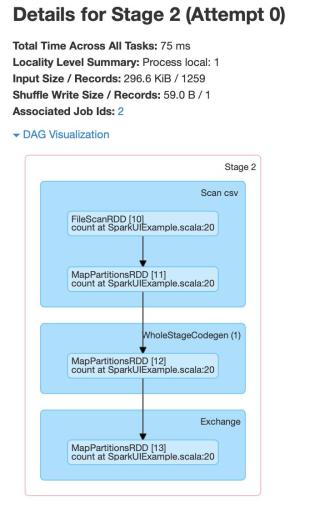
Stage1

Operation in Stage(1) are  
1.FileScanRDD  
2.MapPartitionsRDD  
3.SQLExecutionRDD

As File Scan and MapPartitionsRDD is already explained, let us look at SQLExecutionRDD

**SQLExecutionRDD**

SQLExecutionRDD is Spark property that is used to track multiple Spark jobs that should all together constitute a single structured query execution.

Stage 2

Operation in Stage(2) and Stage(3) are  
1.FileScanRDD  
2.MapPartitionsRDD  
3.WholeStageCodegen  
4.Exchange

**Wholestagecodegen**

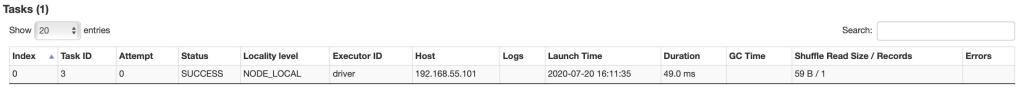
A physical query optimizer in Spark SQL that fuses multiple physical operators

**Exchange**

Exchange is performed because of the COUNT method.  
*As data is divided into partitions and shared among executors, to get count there should be adding of the count of from individual partition.*

Represents the shuffle i.e data movement across the cluster(Executors).  
It is the most expensive operation and if number of partitions is more exchange of data between executors will also be more.

**3. Tasks**

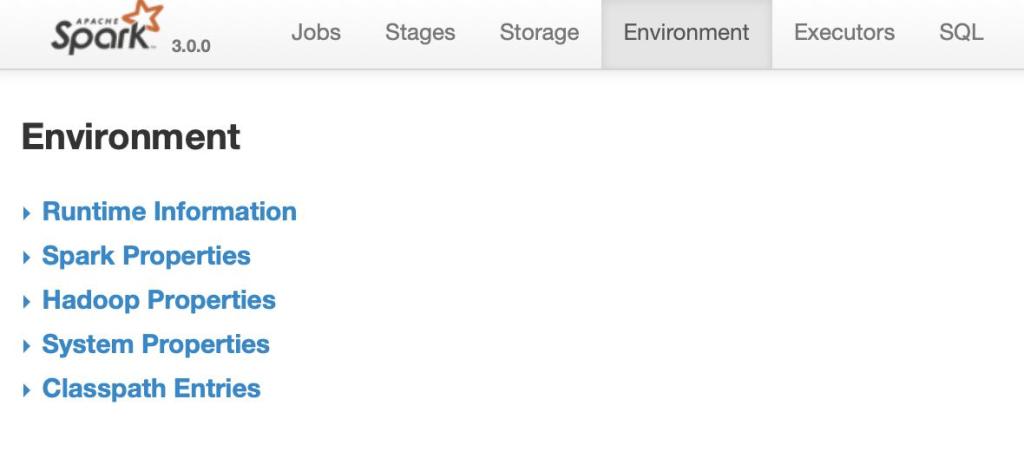
Spark Tasks Tab

Tasks are located at the bottom space in the respective stage.  
Key things to look task page are:  
1. Input Size – Input for the Stage  
2. Shuffle Write-Output is the stage written.

**4. Storage**

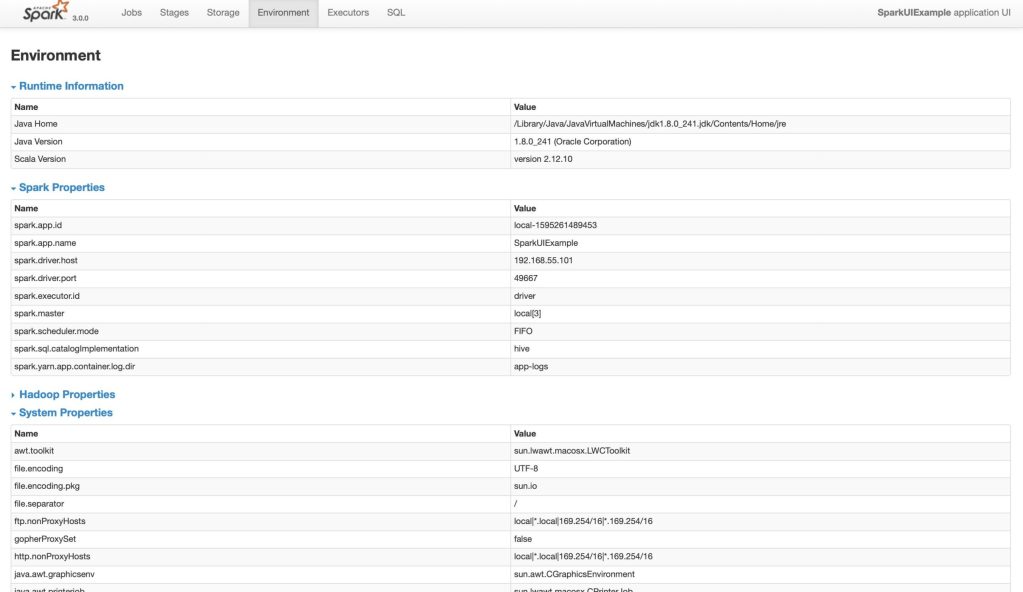
The Storage tab displays the persisted RDDs and DataFrames, if any, in the application. The summary page shows the storage levels, sizes and partitions of all RDDs, and the details page shows the sizes and using executors for all partitions in an RDD or DataFrame.

**5. Environment Tab**

Spark Environment Tab

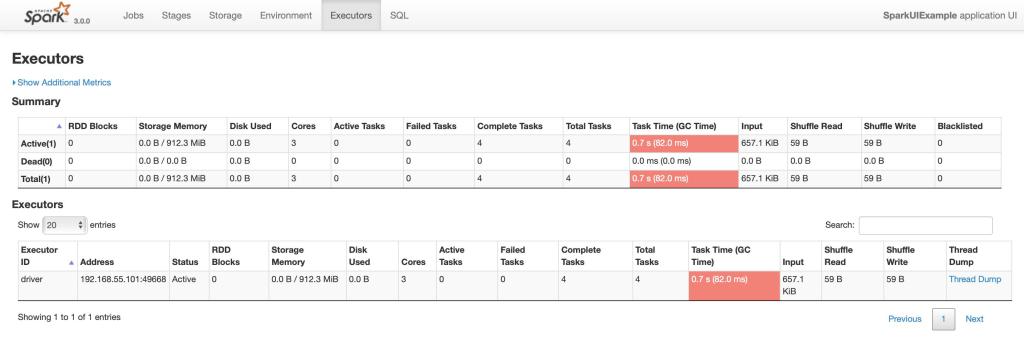
This environment page has five parts. It is a useful place to check whether your properties have been set correctly.

1. **Runtime Information**: simply contains the runtime properties like versions of Java and Scala.
2. **Spark Properties**: lists the application properties like ‘spark.app.name’ and ‘spark.driver.memory’.
3. **Hadoop Properties**: displays properties relative to Hadoop and YARN. **Note**: Properties like [‘](https://spark.apache.org/docs/3.0.0-preview/configuration.html#execution-behavior)spark.hadoop’ are shown not in this part but in ‘Spark Properties’.
4. **System Properties**: shows more details about the JVM.
5. **Classpath Entries**: lists the classes loaded from different sources, which is very useful to resolve class conflicts.

Spark Environment properties

The Environment tab displays the values for the different environment and configuration variables, including JVM, Spark, and system properties.

**6. Executors Tab**

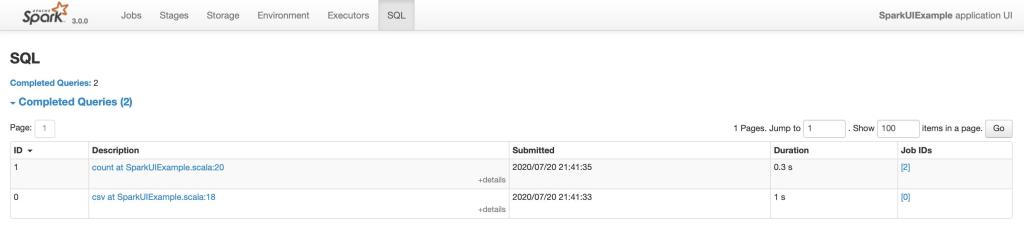
Spark Executors Tab

The Executors tab displays summary information about the executors that were created for the application, including memory and disk usage and task and shuffle information. The Storage Memory column shows the amount of memory used and reserved for caching data.

The Executors tab provides not only resource information like amount of memory, disk, and cores used by each executor but also performance information.

In Executors  
*Number of cores = 3 as I gave master as local with 3 threads*  
*Number of tasks =*4

**7. SQL Tab**

Spark SQL Tab

If the application executes Spark SQL queries then the SQL tab displays information, such as the duration, Spark jobs, and physical and logical plans for the queries.

In our application, we performed read and count operation on files and DataFrame. So both read and count are listed SQL Tab

Some of the resources are gathered from <https://spark.apache.org/> thanks for the information.

## RDD Tutorial

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](https://github.com/spark-examples/pyspark-examples) 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 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)

## 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.

## 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](https://sparkbyexamples.com/pyspark/pyspark-parallelize-create-rdd/) and
* [referencing a dataset in an external storage system](https://sparkbyexamples.com/pyspark/pyspark-read-csv-file-into-dataframe/) (HDFS, S3 and many more).

Before we look into examples, first let’s initialize [SparkSession](https://sparkbyexamples.com/pyspark/pyspark-what-is-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()

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 <a href="https://sparkbyexamples.com/hadoop/how-yarn-works/">yarn (Yet Another Resource Negotiator)</a> 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.

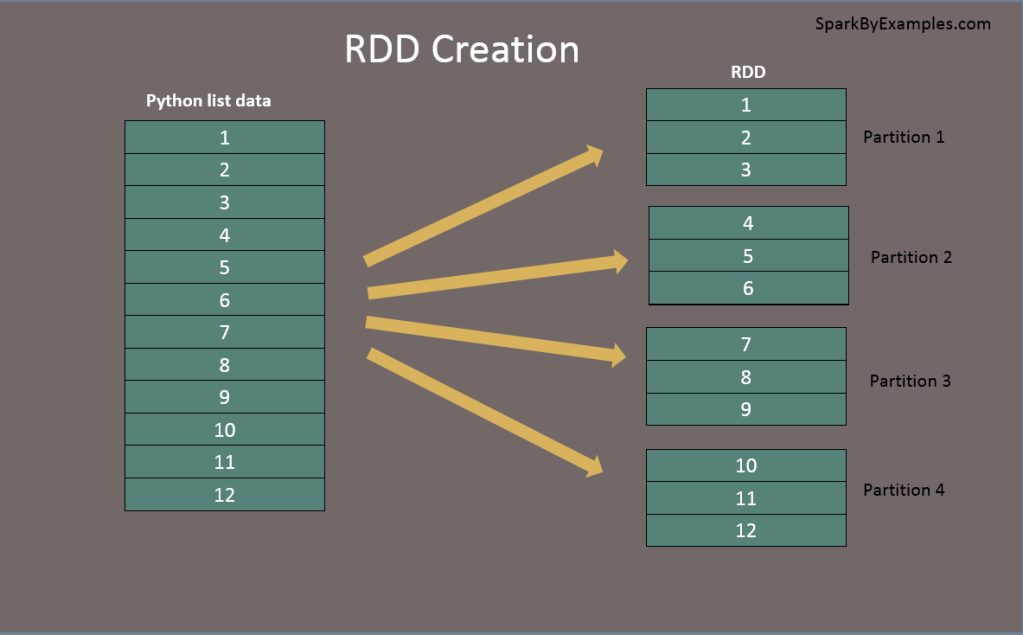
appName() – Used to set your application name.

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

Note: Creating [SparkSession](https://sparkbyexamples.com/pyspark/pyspark-what-is-sparksession/) object, it internally creates one [SparkContext](https://sparkbyexamples.com/pyspark/pyspark-what-is-sparksession/)per JVM.

#### Create RDD using sparkContext.parallelize()

By using parallelize() function of SparkContext ([sparkContext.parallelize()](https://sparkbyexamples.com/pyspark/pyspark-parallelize-create-rdd/) ) 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 used when you already have data in memory that 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)

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](https://sparkbyexamples.com/spark/spark-read-text-file-rdd-dataframe/) (.txt) file into RDD.

#Create RDD from external Data source

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

#### Create RDD using sparkContext.wholeTextFiles()

[wholeTextFiles()](https://sparkbyexamples.com/spark/spark-read-text-file-rdd-dataframe/) function returns a [PairRDD](https://sparkbyexamples.com/apache-spark-rdd/spark-pair-rdd-functions/)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")

Besides using text files, we can also [create RDD from CSV file](https://sparkbyexamples.com/pyspark/pyspark-read-csv-file-into-dataframe/), JSON, and more formats.

#### Create empty RDD using sparkContext.emptyRDD

Using emptyRDD() method on sparkContext we can [create an RDD with no data](https://sparkbyexamples.com/spark/spark-how-to-create-an-empty-rdd/). This method creates an empty RDD with no partition.

# Creates empty RDD with no partition

rdd = spark.sparkContext.emptyRDD

# rddString = spark.sparkContext.emptyRDD[String]

#### Creating empty RDD with partition

Some times 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

## RDD Parallelize

When we use parallelize() or textFile() or wholeTextFiles() methods of [SparkContxt](https://sparkbyexamples.com/pyspark/pyspark-what-is-sparksession/)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

**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

Some times we may need to [repartition the RDD](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/), PySpark provides two ways to repartition; first using repartition() method which shuffles data from all nodes also called full shuffle and second [coalesce()](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/) method which [shuffle](https://sparkbyexamples.com/spark/spark-shuffle-partitions/)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 <a href="https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/">repartition()</a> 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

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

## 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](https://sparkbyexamples.com/pyspark/pyspark-rdd-transformations/) 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](https://github.com/spark-examples/spark-scala-examples/blob/master/src/main/resources/test.txt) project.

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

**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(" "))

**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))

**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)

**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())

**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())

Please refer to this page for the full list of [RDD transformations](https://sparkbyexamples.com/pyspark/pyspark-rdd-transformations/).

### RDD Actions with example

[RDD Action operations](https://sparkbyexamples.com/pyspark/pyspark-rdd-actions/) 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()))

**first**() – Returns the first record.

# Action - first

firstRec = rdd6.first()

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

**max**() – Returns max record.

# Action - max

datMax = rdd6.max()

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

**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]))

**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])

**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])

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

rdd6.saveAsTextFile("/tmp/wordCount")

Note: Please refer to this page for a full list of [RDD actions](https://sparkbyexamples.com/pyspark/pyspark-rdd-actions/).

## Types of RDD

**PairRDDFunctions or PairRDD** – [Pair RDD](https://sparkbyexamples.com/apache-spark-rdd/spark-pair-rdd-functions/) 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](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/) 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](https://sparkbyexamples.com/apache-spark-rdd/different-ways-to-create-spark-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 map 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.

## RDD Persistence Tutorial

PySpark Cache and Persist are optimization techniques to [improve the performance of the RDD](https://sparkbyexamples.com/apache-spark-rdd/spark-rdd-cache-and-persist-example/) 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()](https://sparkbyexamples.com/apache-spark-rdd/spark-rdd-cache-and-persist-example/) 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](https://sparkbyexamples.com/spark/spark-persistance-storage-levels/) **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()

### RDD Persist

PySpark persist() method is used to store the RDD to one of the [storage levels](https://sparkbyexamples.com/spark/spark-persistance-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 **MEMORY\_ONLY** 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)

### 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()

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.

## 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](https://sparkbyexamples.com/pyspark/pyspark-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

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](https://sparkbyexamples.com/pyspark/pyspark-broadcast-variables/) for more detailed example.

### Accumulators

PySpark Accumulators are another type shared variable that are only “added” through an [associative and commutative operation](https://sparkbyexamples.com/spark/spark-accumulators/) 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](https://sparkbyexamples.com/spark/spark-web-ui-understanding/) 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))

PySpark by default provides accumulator methods for long, double and collection types. All these methods are present in [SparkContext](https://sparkbyexamples.com/pyspark/pyspark-what-is-sparksession/) class and return LongAccumulator, DoubleAccumulator, and CollectionAccumulator respectively.

* [Long Accumulator](https://sparkbyexamples.com/spark/spark-accumulators/#LongAccumulator)
* [Double Accumulator](https://sparkbyexamples.com/spark/spark-accumulators/#DoubleAccumulator)
* [Collection Accumulator](https://sparkbyexamples.com/spark/spark-accumulators/#CollectionAccumulator)

# parallelize() – Create RDD from a list data

PySpark parallelize() is a function in SparkContext and is used to create an RDD from a list collection. In this article, I will explain the usage of parallelize to create RDD and how to create an empty RDD with PySpark example.

Before we start let me explain what is RDD, [Resilient Distributed Datasets (**RDD**)](https://sparkbyexamples.com/spark-rdd-tutorial/) is a fundamental data structure of PySpark, It is an immutable distributed collection of objects. Each dataset in **RDD** is divided into logical partitions, which may be computed on different nodes of the cluster.

* PySpark Parallelizing an existing collection in your driver program.

Below is an example of how to create an RDD using a parallelize method from [Sparkcontext](https://sparkbyexamples.com/spark/spark-sparkcontext/). sparkContext.parallelize([1,2,3,4,5,6,7,8,9,10]) creates an RDD with a list of Integers.

## ****Using sc.parallelize on PySpark Shell or REPL****

PySpark shell provides [SparkContext](https://sparkbyexamples.com/spark/spark-sparkcontext/) variable “sc”, use sc.parallelize() to create an RDD.

rdd = sc.parallelize([1,2,3,4,5,6,7,8,9,10])

## ****Using PySpark sparkContext.parallelize****() in application

Since PySpark 2.0, First, you need to create a [SparkSession](https://sparkbyexamples.com/pyspark/pyspark-what-is-sparksession/) which internally creates a SparkContext for you.

import pyspark

from pyspark.sql import SparkSession

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

sparkContext=spark.sparkContext

Now, use sparkContext.parallelize() to create rdd from a list or collection.

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

rddCollect = rdd.collect()

print("Number of Partitions: "+str(rdd.getNumPartitions()))

print("Action: First element: "+str(rdd.first()))

print(rddCollect)

By executing the above program you should see below output.

Number of Partitions: 4

Action: First element: 1

[1, 2, 3, 4, 5]

parallelize() function also has another signature which additionally takes integer argument to specifies the number of partitions. Partitions are basic units of parallelism in PySpark.

Remember, RDDs in PySpark are a collection of partitions.

## create empty RDD by using ****sparkContext.parallelize****

Some times we may need to create empty RDD and you can also use parallelize() in order to create it.

emptyRDD = sparkContext.emptyRDD()

emptyRDD2 = rdd=sparkContext.parallelize([])

print("is Empty RDD : "+str(emptyRDD2.isEmpty()))

The complete code can be downloaded from [GitHub – PySpark Examples project](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-parallelize.py)

# Repartition() vs Coalesce()

Let’s see the difference between PySpark repartition() vs coalesce(), repartition() is used to increase or decrease the RDD/DataFrame partitions whereas the PySpark coalesce() is used to only decrease the number of partitions in an efficient way.

In this article, you will learn what is PySpark repartition() and coalesce() methods? and the difference between repartition vs coalesce with PySpark examples.

* [RDD Partition](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/#rdd-partition)
  + [RDD repartition](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/#rdd-repartition)
  + [RDD coalesce](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/#rdd-coalesce)
* [DataFrame Partition](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/#dataframe-partition)
  + [DataFrame repartition](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/#dataframe-repartition)
  + [DataFrame coalesce](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/#dataframe-%20coalesce)

One important point to note is, PySpark repartition() and coalesce() are **very expensive operations** as they **shuffle the data across many partitions** hence try to minimize using these as much as possible.

## RDD Repartition() vs Coalesce()

In RDD, you can create parallelism at the time of the [creation of an RDD](https://sparkbyexamples.com/apache-spark-rdd/different-ways-to-create-spark-rdd/) using [parallelize()](https://sparkbyexamples.com/apache-spark-rdd/how-to-create-an-rdd-using-parallelize/), [textFile()](https://sparkbyexamples.com/apache-spark-rdd/spark-read-multiple-text-files-into-a-single-rdd/) and [wholeTextFiles()](https://sparkbyexamples.com/apache-spark-rdd/spark-read-multiple-text-files-into-a-single-rdd/).

rdd = spark.sparkContext.parallelize((0,20))

print("From local[5]"+str(rdd.getNumPartitions()))

rdd1 = spark.sparkContext.parallelize((0,25), 6)

print("parallelize : "+str(rdd1.getNumPartitions()))

rddFromFile = spark.sparkContext.textFile("src/main/resources/test.txt",10)

print("TextFile : "+str(rddFromFile.getNumPartitions()))

The above example yields below output.

From local[5] : 5

Parallelize : 6

TextFile : 10

spark.sparkContext.parallelize(Range(0,20),6) distributes RDD into 6 partitions and the data is distributed as below.

rdd1.saveAsTextFile("/tmp/partition")

//Writes 6 part files, one for each partition

Partition 1 : 0 1 2

Partition 2 : 3 4 5

Partition 3 : 6 7 8 9

Partition 4 : 10 11 12

Partition 5 : 13 14 15

Partition 6 : 16 17 18 19

### 1.1 RDD repartition()

Spark RDD repartition() method is used to increase or decrease the partitions. The below example decreases the partitions from 10 to 4 by moving data from all partitions.

rdd2 = rdd1.repartition(4)

print("Repartition size : "+str(rdd2.getNumPartitions()))

rdd2.saveAsTextFile("/tmp/re-partition")

This yields output Repartition size : 4 and the repartition re-distributes the data(as shown below) from all partitions which is full shuffle leading to very expensive operation when dealing with billions and trillions of data.

Partition 1 : 1 6 10 15 19

Partition 2 : 2 3 7 11 16

Partition 3 : 4 8 12 13 17

Partition 4 : 0 5 9 14 18

### 1.2 RDD coalesce()

Spark RDD coalesce() is used only to reduce the number of partitions. This is optimized or improved version of repartition() where the movement of the data across the partitions is lower using coalesce.

rdd3 = rdd1.coalesce(4)

print("Repartition size : "+str(rdd3.getNumPartitions()))

rdd3.saveAsTextFile("/tmp/coalesce")

If you compared the below output with section 1, you will notice partition 3 has been moved to 2 and Partition 6 has moved to 5, resulting data movement from just 2 partitions.

Partition 1 : 0 1 2

Partition 2 : 3 4 5 6 7 8 9

Partition 4 : 10 11 12

Partition 5 : 13 14 15 16 17 18 19

### 1.3 Complete Example of PySpark RDD repartition and coalesce

Below is a complete example of PySpark RDD repartition and coalesce in Scala language.

import pyspark

from pyspark.sql import SparkSession

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

.master("local[5]").getOrCreate()

df = spark.range(0,20)

print(df.rdd.getNumPartitions())

spark.conf.set("spark.sql.shuffle.partitions", "500")

rdd = spark.sparkContext.parallelize((0,20))

print("From local[5]"+str(rdd.getNumPartitions()))

rdd1 = spark.sparkContext.parallelize((0,25), 6)

print("parallelize : "+str(rdd1.getNumPartitions()))

"""rddFromFile = spark.sparkContext.textFile("src/main/resources/test.txt",10)

print("TextFile : "+str(rddFromFile.getNumPartitions())) """

rdd1.saveAsTextFile("c://tmp/partition2")

rdd2 = rdd1.repartition(4)

print("Repartition size : "+str(rdd2.getNumPartitions()))

rdd2.saveAsTextFile("c://tmp/re-partition2")

rdd3 = rdd1.coalesce(4)

print("Repartition size : "+str(rdd3.getNumPartitions()))

rdd3.saveAsTextFile("c:/tmp/coalesce2")

## DataFrame repartition() vs coalesce()

Like RDD, you can’t specify the partition/parallelism while [creating DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/). DataFrame by default internally uses the methods specified in Section 1 to determine the default partition and splits the data for parallelism.

If you are not familiar with DataFrame, I will recommend to learn the DataFrame before proceeding further on this article.

import pyspark

from pyspark.sql import SparkSession

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

.master("local[5]").getOrCreate()

df=spark.range(0,20)

print(df.rdd.getNumPartitions())

df.write.mode("overwrite").csv("c:/tmp/partition.csv")

The above example creates 5 partitions as specified in master("local[5]") and the data is distributed across all these 5 partitions.

Partition 1 : 0 1 2 3

Partition 2 : 4 5 6 7

Partition 3 : 8 9 10 11

Partition 4 : 12 13 14 15

Partition 5 : 16 17 18 19

### 2.1 DataFrame repartition()

Similar to RDD, the PySpark DataFrame repartition() method is used to increase or decrease the partitions. The below example increases the partitions from 5 to 6 by moving data from all partitions.

df2 = df.repartition(6)

print(df2.rdd.getNumPartitions())

Just increasing 1 partition results data movements from all partitions.

Partition 1 : 14 1 5

Partition 2 : 4 16 15

Partition 3 : 8 3 18

Partition 4 : 12 2 19

Partition 5 : 6 17 7 0

Partition 6 : 9 10 11 13

And, even decreasing the partitions also results in moving data from all partitions. hence when you wanted to decrease the partition recommendation is to use coalesce()/

### 2.2 DataFrame coalesce()

Spark DataFrame coalesce() is used only to decrease the number of partitions. This is an optimized or improved version of repartition() where the movement of the data across the partitions is fewer using coalesce.

df3 = df.coalesce(2)

print(df3.rdd.getNumPartitions())

This yields output 2 and the resultant partition looks like

Partition 1 : 0 1 2 3 8 9 10 11

Partition 2 : 4 5 6 7 12 13 14 15 16 17 18 19

Since we are reducing 5 to 2 partitions, the data movement happens only from 3 partitions and it moves to remain 2 partitions.

### Default Shuffle Partition

Calling groupBy(),union(),join() and similar functions on DataFrame results in shuffling data between multiple executors and even machines and finally repartitions data into **200** partitions by default. PySpark default defines shuffling partition to 200 using spark.sql.shuffle.partitions configuration.

df4 = df.groupBy("id").count()

print(df4.rdd.getNumPartitions())

Post shuffle operations, you can change the partitions either using coalesce() or repartition().

#### Conclusion

In this PySpark repartition() vs coalesce() article, you have learned how to create an RDD with partition, repartition the RDD using coalesce(), repartition DataFrame using repartition() and coalesce() methods and leaned the difference between repartition and coalesce.

# Broadcast Variables

In PySpark RDD and DataFrame, 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 workers using efficient broadcast algorithms to reduce communication costs.

### Use case

Let me explain with an example when to use broadcast variables, assume you are getting a two-letter country state code in a file and you wanted to transform it to full state name, (for example CA to California, NY to New York e.t.c) by doing a lookup to reference mapping. In some instances, this data could be large and you may have many such lookups (like zip code e.t.c).

Instead of distributing this information along with each task over the network (overhead and time consuming), we can use the broadcast variable to cache this lookup info on each machine and tasks use this cached info while executing the transformations.

## How does PySpark Broadcast work?

Broadcast variables are used in the same way for RDD, DataFrame.

When you run a PySpark RDD, DataFrame applications that have 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
* Spark 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.

You should be creating and using broadcast variables for data that shared across multiple stages and tasks.

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.

## How to create Broadcast variable

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.

### In PySpark shell

broadcastVar = sc.broadcast(Array(0, 1, 2, 3))

broadcastVar.value

## PySpark RDD Broadcast variable example

Below is a very simple example of how to use broadcast variables on RDD. This example defines commonly used data (states) in a Map variable and distributes the variable using SparkContext.broadcast() and then use these variables on RDD map() transformation.

import pyspark

from pyspark.sql import SparkSession

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

states = {"NY":"New York", "CA":"California", "FL":"Florida"}

broadcastStates = spark.sparkContext.broadcast(states)

data = [("James","Smith","USA","CA"),

("Michael","Rose","USA","NY"),

("Robert","Williams","USA","CA"),

("Maria","Jones","USA","FL")

]

rdd = spark.sparkContext.parallelize(data)

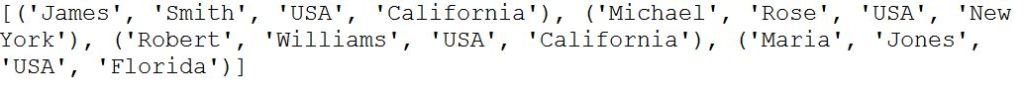
def state\_convert(code):

return broadcastStates.value[code]

result = rdd.map(lambda x: (x[0],x[1],x[2],state\_convert(x[3]))).collect()

print(result)

Yields below output



## PySpark DataFrame Broadcast variable example

Below is an example of how to use broadcast variables on DataFrame, similar to above RDD example, This also uses commonly used data (states) in a Map variable and distributes the variable using SparkContext.broadcast() and then use these variables on DataFrame map() transformation.

If you are not familiar with DataFrame, I will recommend to learn the DataFrame before proceeding further on this article.

import pyspark

from pyspark.sql import SparkSession

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

states = {"NY":"New York", "CA":"California", "FL":"Florida"}

broadcastStates = spark.sparkContext.broadcast(states)

data = [("James","Smith","USA","CA"),

("Michael","Rose","USA","NY"),

("Robert","Williams","USA","CA"),

("Maria","Jones","USA","FL")

]

columns = ["firstname","lastname","country","state"]

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

df.printSchema()

df.show(truncate=False)

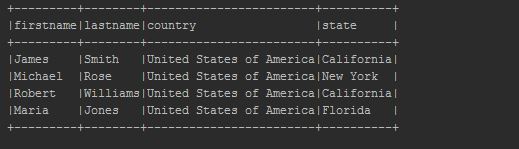
def state\_convert(code):

return broadcastStates.value[code]

result = df.rdd.map(lambda x: (x[0],x[1],x[2],state\_convert(x[3]))).toDF(columns)

result.show(truncate=False)

Above example first [creates a DataFrame](https://sparkbyexamples.com/spark/different-ways-to-create-a-spark-dataframe/), transform the data using broadcast variable and yields below output.



You can also use the broadcast variable on the filter and joins. Below is a filter example.

# Broadcast variable on filter

filteDf= df.where((df['state'].isin(broadcastStates.value)))

### Conclusion

In this PySpark Broadcast variable article, you have learned what is Broadcast variable, it’s advantage and how to use in RDD and Dataframe with Pyspark example.

### Reference

* <https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#broadcast-variables>

# PySpark Accumulator with Example

The PySpark Accumulator is a shared variable that is used with RDD and DataFrame to perform sum and counter operations similar to Map-reduce counters. These variables are shared by all executors to update and add information through aggregation or computative operations.

In this article, I’ve explained what is PySpark Accumulator, how to create, and using it on RDD and DataFrame with an example.

**What is PySpark Accumulator?**

Accumulators are write-only and initialize once variables where only tasks that are running on workers are allowed to update and updates from the workers get propagated automatically to the driver program. But, only the driver program is allowed to access the Accumulator variable using the **value**property.

**How to create Accumulator variable in PySpark?**

Using **accumulator**() from SparkContext class we can create an Accumulator in PySpark programming. Users can also create Accumulators for custom types using AccumulatorParam class of PySpark.

**Some points to note..**

* **sparkContext.accumulator()**is used to define accumulator variables.
* **add()** function is used to add/update a value in accumulator
* **value** property on the accumulator variable is used to retrieve the value from the accumulator.

We can create Accumulators in PySpark for primitive types **int** and **float**. Users can also create Accumulators for custom types using AccumulatorParam class of PySpark.

## Creating Accumulator Variable

Below is an example of how to create an accumulator variable “**accum**” of type int and using it to sum all values in an RDD.

accum=sc.accumulator(0)

rdd=spark.sparkContext.parallelize([1,2,3,4,5])

rdd.foreach(lambda x:accum.add(x))

print(accum.value) #Accessed by driver

Here, we have created an accumulator variable **accum** using **spark.sparkContext.accumulator(0)** with initial value 0. Later, we are [iterating each element in an rdd using foreach() action](https://sparkbyexamples.com/spark/spark-foreach-usage-with-examples/) and adding each element of rdd to accum variable. Finally, we are getting accumulator value using **accum.value**property.

Note that, In this example, rdd.foreach() is executed on workers and accum.value is called from PySpark driver program.

Let’s see another example of an accumulator, this time will do with a function.

accuSum=spark.sparkContext.accumulator(0)

def countFun(x):

global accuSum

accuSum+=x

rdd.foreach(countFun)

print(accuSum.value)

We can also use accumulators to do a counters.

accumCount=spark.sparkContext.accumulator(0)

rdd2=spark.sparkContext.parallelize([1,2,3,4,5])

rdd2.foreach(lambda x:accumCount.add(1))

print(accumCount.value)

## Accumulator Example

Below is a complete RDD example of using different accumulators that I was able to run on my environment.

import pyspark

from pyspark.sql import SparkSession

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

accum=spark.sparkContext.accumulator(0)

rdd=spark.sparkContext.parallelize([1,2,3,4,5])

rdd.foreach(lambda x:accum.add(x))

print(accum.value)

accuSum=spark.sparkContext.accumulator(0)

def countFun(x):

global accuSum

accuSum+=x

rdd.foreach(countFun)

print(accuSum.value)

accumCount=spark.sparkContext.accumulator(0)

rdd2=spark.sparkContext.parallelize([1,2,3,4,5])

rdd2.foreach(lambda x:accumCount.add(1))

print(accumCount.value)

#### Conclusion

In summary, PySpark Accumulators are shared variables that can be updated by executors and propagates back to driver program. These variables are used to add sum or counts and final results can be accessed only by driver program.

#### Reference

* <https://spark.apache.org/docs/latest/api/python/_modules/pyspark/accumulators.html>

# Create DataFrame with Examples

You can manually c**reate 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**](https://sparkbyexamples.com/spark/spark-fetch-more-than-20-rows-full-column-value/)

# Show 50 rows

df.show(50)

# Show 20 rows with full column value

df.show(truncate=False)

# Show 50 rows & full column value

df.show(50,truncate=False)

# Show 20 rows, column length 20 & displays data in vertical

**df.show(n=20,truncate=20,vertical=True)**

* [**Get Current Number of Partitions of Spark DataFrame**](https://sparkbyexamples.com/spark/spark-get-current-number-of-partitions-of-dataframe/)

# RDD

rdd.getNumPartitions()

# For DataFrame, convert to RDD first

df.rdd.getNumPartitions()

* [**How to check if Column Present in Spark DataFrame**](https://sparkbyexamples.com/spark/spark-check-column-present-in-dataframe/)

**df.columns.map(\_.toUpperCase).contains(columnNameToCheck**.toUpperCase)

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

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

| 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")]

## 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](https://sparkbyexamples.com/spark/different-ways-to-create-spark-rdd/) from a collection List by calling [parallelize()](https://sparkbyexamples.com/pyspark/pyspark-parallelize-create-rdd/) function from [SparkContext](https://sparkbyexamples.com/spark/spark-sparkcontext/). We would need this **rdd** object for all our examples below.

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

rdd = spark.sparkContext.parallelize(data)

### 1.1 Using toDF() function

PySpark RDD’s toDF() method is used to create a DataFrame from 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()

printschema() yields the below output.

root

|-- \_1: string (nullable = true)

|-- \_2: string (nullable = true)

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()

This yields schema of the DataFrame with column names.

root

|-- language: string (nullable = true)

|-- users: string (nullable = true)

By default, the datatype of these columns infers to the type of data. We can change this behavior by [supplying schema](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/), 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](https://sparkbyexamples.com/tag/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)

## 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)

### 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)

### 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) \

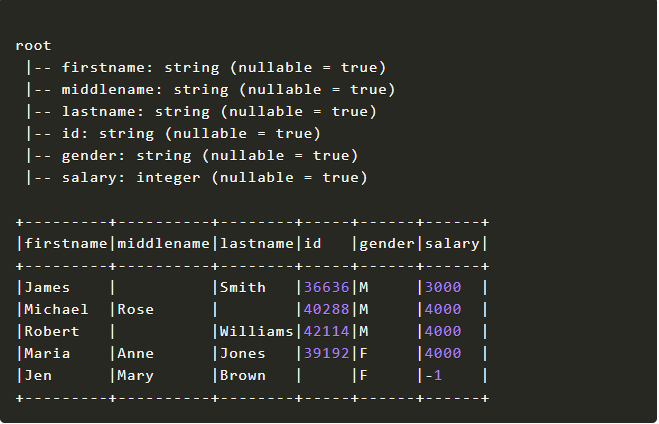
])

df = spark.createDataFrame(data=data2,schema=schema)

df.printSchema()

df.show(truncate=False)

This yields below output.



## 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](https://sparkbyexamples.com/pyspark/pyspark-read-csv-file-into-dataframe/)

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

### 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")

### 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")

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](https://sparkbyexamples.com/pyspark/pyspark-read-and-write-parquet-file/)
* [DataFrame from Avro source](https://sparkbyexamples.com/spark/using-avro-data-files-from-spark-sql-2-4/)
* [DataFrame by Streaming data from Kafka](https://sparkbyexamples.com/spark/spark-streaming-kafka-consumer-example-in-json-format/)

# Create an Empty DataFrame & RDD

In this article, I will explain how to create an empty PySpark DataFrame/RDD manually with or without schema (column names) in different ways. Below I have explained one of the many scenarios where we need to create an empty DataFrame.

While working with files, sometimes we may not receive a file for processing, however, we still need to create a DataFrame manually with the same schema we expect. If we don’t create with the same schema, our operations/transformations (like union’s) on DataFrame fail as we refer to the columns that may not present.

To handle situations similar to these, we always need to create a DataFrame with the same schema, which means the same column names and datatypes regardless of the file exists or empty file processing.

## 1. Create Empty RDD in PySpark

Create an empty RDD by using emptyRDD() of SparkContext for example spark.sparkContext.emptyRDD().

from pyspark.sql import SparkSession

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

#Creates Empty RDD

emptyRDD = spark.sparkContext.emptyRDD()

print(emptyRDD)

#Diplays

#EmptyRDD[188] at emptyRDD

Alternatively you can also get empty RDD by using spark.sparkContext.parallelize([]).

#Creates Empty RDD using parallelize

rdd2= spark.sparkContext.parallelize([])

print(rdd2)

#EmptyRDD[205] at emptyRDD at NativeMethodAccessorImpl.java:0

#ParallelCollectionRDD[206] at readRDDFromFile at PythonRDD.scala:262

**Note:** If you try to perform operations on empty RDD you going to get ValueError("RDD is empty").

## 2. Create Empty DataFrame with Schema (StructType)

In order to create an empty PySpark DataFrame manually with schema ( column names & data types) first, [Create a schema using StructType and StructField](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/) .

#Create Schema

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

schema = StructType([

StructField('firstname', StringType(), True),

StructField('middlename', StringType(), True),

StructField('lastname', StringType(), True)

])

Now use the empty RDD created above and pass it to createDataFrame() of [SparkSession](https://sparkbyexamples.com/pyspark/pyspark-what-is-sparksession/) along with the schema for column names & data types.

#Create empty DataFrame from empty RDD

df = spark.createDataFrame(emptyRDD,schema)

df.printSchema()

This yields below schema of the empty DataFrame.

root

|-- firstname: string (nullable = true)

|-- middlename: string (nullable = true)

|-- lastname: string (nullable = true)

## 3. Convert Empty RDD to DataFrame

You can also create empty DataFrame by converting empty RDD to DataFrame using toDF().

#Convert empty RDD to Dataframe

df1 = emptyRDD.toDF(schema)

df1.printSchema()

## 4. Create Empty DataFrame with Schema.

So far I have covered creating an empty DataFrame from RDD, but here will create it manually with schema and without RDD.

#Create empty DataFrame directly.

df2 = spark.createDataFrame([], schema)

df2.printSchema()

## 5. Create Empty DataFrame without Schema (no columns)

To create empty DataFrame with out schema (no columns) just create a empty schema and use it while creating PySpark DataFrame.

#Create empty DatFrame with no schema (no columns)

df3 = spark.createDataFrame([], StructType([]))

df3.printSchema()

#print below empty schema

#root

# Convert PySpark RDD to DataFrame

In PySpark, toDF() function of the RDD is used to convert RDD to DataFrame. We would need to convert RDD to DataFrame as DataFrame provides more advantages over RDD. For instance, DataFrame is a distributed collection of data organized into named columns similar to Database tables and provides optimization and performance improvements.

## 1. Create PySpark RDD

First, let’s create an RDD by passing Python list object to sparkContext.parallelize() function. We would need this rdd object for all our examples below.

In PySpark, when you have data in a list meaning you have a collection of data in a PySpark driver memory when you create an RDD, this collection is going to be [parallelized](https://sparkbyexamples.com/pyspark/pyspark-parallelize-create-rdd-from-collection/).

from pyspark.sql import SparkSession

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

dept = [("Finance",10),("Marketing",20),("Sales",30),("IT",40)]

rdd = spark.sparkContext.parallelize(dept)

## 2. Convert PySpark RDD to DataFrame

Converting PySpark RDD to DataFrame can be done using toDF(), createDataFrame(). In this section, I will explain these two methods.

### 2.1 Using rdd.toDF() function

PySpark provides toDF() function in RDD which can be used to convert RDD into Dataframe

df = rdd.toDF()

df.printSchema()

df.show(truncate=False)

By default, toDF() function creates column names as “\_1” and “\_2”. This snippet yields below schema.

root

|-- \_1: string (nullable = true)

|-- \_2: long (nullable = true)

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

|\_1 |\_2 |

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

|Finance |10 |

|Marketing|20 |

|Sales |30 |

|IT |40 |

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

toDF() has another signature that takes arguments to define column names as shown below.

deptColumns = ["dept\_name","dept\_id"]

df2 = rdd.toDF(deptColumns)

df2.printSchema()

df2.show(truncate=False)

Outputs below schema.

root

|-- dept\_name: string (nullable = true)

|-- dept\_id: long (nullable = true)

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

|dept\_name|dept\_id|

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

|Finance |10 |

|Marketing|20 |

|Sales |30 |

|IT |40 |

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

### 2.2 Using PySpark createDataFrame() function

SparkSession class provides createDataFrame() method to create DataFrame and it takes rdd object as an argument.

deptDF = spark.createDataFrame(rdd, schema = deptColumns)

deptDF.printSchema()

deptDF.show(truncate=False)

This yields the same output as above.

## 2.3 Using createDataFrame() with StructType schema

When you infer the schema, by default the datatype of the columns is derived from the data and set’s nullable to true for all columns. We can change this behavior by supplying [schema](https://sparkbyexamples.com/spark/spark-schema-explained-with-examples/) using [StructType](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/) – where we can specify a column name, data type and nullable for each field/column.

If you wanted to know more about StructType, please go through [how to use StructType and StructField to define the custom schema](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/).

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

deptSchema = StructType([

StructField('dept\_name', StringType(), True),

StructField('dept\_id', StringType(), True)

])

deptDF1 = spark.createDataFrame(rdd, schema = deptSchema)

deptDF1.printSchema()

deptDF1.show(truncate=False)

This also yields the same output.

## 3. Complete Example

import pyspark

from pyspark.sql import SparkSession

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

dept = [("Finance",10),("Marketing",20),("Sales",30),("IT",40)]

rdd = spark.sparkContext.parallelize(dept)

df = rdd.toDF()

df.printSchema()

df.show(truncate=False)

deptColumns = ["dept\_name","dept\_id"]

df2 = rdd.toDF(deptColumns)

df2.printSchema()

df2.show(truncate=False)

deptDF = spark.createDataFrame(rdd, schema = deptColumns)

deptDF.printSchema()

deptDF.show(truncate=False)

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

deptSchema = StructType([

StructField('dept\_name', StringType(), True),

StructField('dept\_id', StringType(), True)

])

deptDF1 = spark.createDataFrame(rdd, schema = deptSchema)

deptDF1.printSchema()

deptDF1.show(truncate=False)

The complete code can be downloaded from [GitHub](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-rdd-to-dataframe.py)

## 4. Conclusion:

In this article, you have learned how to convert PySpark RDD to DataFrame, we would need these frequently while working in PySpark as these provides optimization and performance over RDD

# Convert PySpark DataFrame to Pandas

(Spark with Python)PySpark DataFrame can be converted to Python Pandas DataFrame using a function toPandas(), In this article, I will explain how to create Pandas DataFrame from PySpark (Spark) Dataframe with examples.

Before we start first understand the main differences between the Pandas & PySpark, operations on Pyspark run faster than Pandas due to its distributed nature and parallel execution on multiple cores and machines.

In other words, pandas run operations on a single node whereas PySpark runs on multiple machines. If you are working on a Machine Learning application where you are dealing with larger datasets, PySpark processes operations many times faster than pandas.

After processing data in PySpark we would need to convert it back to Pandas DataFrame for a further procession with Machine Learning application or any Python applications.

## Prepare PySpark DataFrame

In order to explain with an example first let’s [create a PySpark DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/).

import pyspark

from pyspark.sql import SparkSession

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

data = [("James","","Smith","36636","M",60000),

("Michael","Rose","","40288","M",70000),

("Robert","","Williams","42114","",400000),

("Maria","Anne","Jones","39192","F",500000),

("Jen","Mary","Brown","","F",0)]

columns = ["first\_name","middle\_name","last\_name","dob","gender","salary"]

pysparkDF = spark.createDataFrame(data = data, schema = columns)

pysparkDF.printSchema()

pysparkDF.show(truncate=False)

This yields below schema and result of the DataFrame.

root

|-- first\_name: string (nullable = true)

|-- middle\_name: string (nullable = true)

|-- last\_name: string (nullable = true)

|-- dob: string (nullable = true)

|-- gender: string (nullable = true)

|-- salary: long (nullable = true)

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

|first\_name|middle\_name|last\_name|dob |gender|salary|

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

|James | |Smith |36636|M |60000 |

|Michael |Rose | |40288|M |70000 |

|Robert | |Williams |42114| |400000|

|Maria |Anne |Jones |39192|F |500000|

|Jen |Mary |Brown | |F |0 |

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

## Convert PySpark Dataframe to Pandas DataFrame

PySpark DataFrame provides a method toPandas() to convert it Python Pandas DataFrame.

toPandas() results in the collection of all records in the PySpark DataFrame to the driver program and should be done on a small subset of the data. running on larger dataset’s results in memory error and crashes the application.

pandasDF = pysparkDF.toPandas()

print(pandasDF)

This yields the below panda’s dataframe. Note that pandas add a sequence number to the result.

first\_name middle\_name last\_name dob gender salary

0 James Smith 36636 M 60000

1 Michael Rose 40288 M 70000

2 Robert Williams 42114 400000

3 Maria Anne Jones 39192 F 500000

4 Jen Mary Brown F 0

## Convert Spark Nested Struct DataFrame to Pandas

Most of the time data in PySpark DataFrame will be in a structured format meaning one column contains other columns so let’s see how it convert to Pandas. Here is an example with nested struct where we have firstname, middlename and lastname are part of the name column.

# Nested structure elements

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

dataStruct = [(("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") \

]

schemaStruct = StructType([

StructField('name', StructType([

StructField('firstname', StringType(), True),

StructField('middlename', StringType(), True),

StructField('lastname', StringType(), True)

])),

StructField('dob', StringType(), True),

StructField('gender', StringType(), True),

StructField('salary', StringType(), True)

])

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

df.printSchema()

pandasDF2 = df.toPandas()

print(pandasDF2)

Converting structured DataFrame to Pandas DataFrame results below output.

name dob gender salary

0 (James, , Smith) 36636 M 3000

1 (Michael, Rose, ) 40288 M 4000

2 (Robert, , Williams) 42114 M 4000

3 (Maria, Anne, Jones) 39192 F 4000

4 (Jen, Mary, Brown) F -1

#### Conclusion

In this simple article, you have learned to convert Spark DataFrame to pandas using toPandas() function of the Spark DataFrame. also have seen a similar example with complex nested structure elements. toPandas() results in the collection of all records in the DataFrame to the driver program and should be done on a small subset of the data.

Happy Learning !!

Reference: <https://docs.databricks.com/spark/latest/spark-sql/spark-pandas.html>

# PySpark show() – Display DataFrame Contents in Table

PySpark DataFrame show() is used to display the contents of the DataFrame in a Table Row & Column Format. By default, it shows only 20 Rows, and the column values are truncated at 20 characters.

## 1. PySpark DataFrame show() Syntax & Example

### 1.1 Syntax

def show(self, n=20, truncate=True, vertical=False):

## 1.2 Example

from pyspark.sql import SparkSession

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

columns = ["Seqno","Quote"]

data = [("1", "Be the change that you wish to see in the world"),

("2", "Everyone thinks of changing the world, but no one thinks of changing himself."),

("3", "The purpose of our lives is to be happy."),

("4", "Be cool.")]

df = spark.createDataFrame(data,columns)

df.show()

#+-----+--------------------+

#|Seqno| Quote|

#+-----+--------------------+

#| 1|Be the change tha...|

#| 2|Everyone thinks o...|

#| 3|The purpose of ou...|

#| 4| Be cool.|

#+-----+--------------------+

As you see above, values in the Quote column is truncated at 20 characters, Let’s see how to display the full column contents.

#Display full column contents

df.show(truncate=False)

#+-----+-----------------------------------------------------------------------------+

#|Seqno|Quote |

#+-----+-----------------------------------------------------------------------------+

#|1 |Be the change that you wish to see in the world |

#|2 |Everyone thinks of changing the world, but no one thinks of changing himself.|

#|3 |The purpose of our lives is to be happy. |

#|4 |Be cool. |

#+-----+-----------------------------------------------------------------------------+

By default show() method displays only 20 rows from PySpark DataFrame. The below example limit the rows to 2 and full column contents. Our DataFrame has just 4 rows hence I can’t demonstrate with more than 4 rows. If you have a DataFrame with thousands of rows try changing the value from 2 to 100 to display more than 20 rows.

# Display 2 rows and full column contents

df.show(2,truncate=False)

#+-----+-----------------------------------------------------------------------------+

#|Seqno|Quote |

#+-----+-----------------------------------------------------------------------------+

#|1 |Be the change that you wish to see in the world |

#|2 |Everyone thinks of changing the world, but no one thinks of changing himself.|

#+-----+-----------------------------------------------------------------------------+

You can also truncate the column value at desired length.

# Display 2 rows & column values 25 characters

df.show(2,truncate=25)

#+-----+-------------------------+

#|Seqno| Quote|

#+-----+-------------------------+

#| 1|Be the change that you...|

#| 2|Everyone thinks of cha...|

#+-----+-------------------------+

#only showing top 2 rows

Finally, let’s see how to display the DataFrame vertically record by record.

# Display DataFrame rows & columns vertically

df.show(n=3,truncate=25,vertical=True)

#-RECORD 0--------------------------

# Seqno | 1

# Quote | Be the change that you...

#-RECORD 1--------------------------

# Seqno | 2

# Quote | Everyone thinks of cha...

#-RECORD 2--------------------------

# Seqno | 3

# Quote | The purpose of our liv...

# PySpark StructType & StructField Explained with Examples

PySpark StructType & StructField classes are used to programmatically specify the schema to the DataFrame and creating complex columns like nested struct, array and map columns. [StructType](https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/types/StructType.scala) is a collection of [StructField’s](https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/types/StructField.scala) that defines column name, column data type, boolean to specify if the field can be nullable or not and metadata.

In this article, I will explain different ways to define the structure of DataFrame using StructType with PySpark examples. Though PySpark infers a schema from data, some times we may need to define our own column names and data types and this article explains how to define simple, nested, and complex schemas.

## 1. StructType – Defines the structure of the Dataframe

PySpark provides from pyspark.sql.types import StructType class to define the structure of the DataFrame.

StructType is a collection or list of StructField objects.

printSchema() method on the DataFrame shows StructType columns as “struct”.

## 2. StructField – Defines the metadata of the DataFrame column

PySpark provides pyspark.sql.types import StructField class to define the columns which includes column name(String), column type ([DataType](https://sparkbyexamples.com/spark/spark-sql-dataframe-data-types/)), nullable column (Boolean) and metadata (MetaData)

## 3. Using PySpark StructType & StructField with DataFrame

While [creating a PySpark DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/) we can specify the structure using StructType and StructField classes. As specified in the introduction, StructType is a collection of StructField’s which is used to define the column name, data type, and a flag for nullable or not. Using StructField we can also add nested struct schema, [ArrayType](https://sparkbyexamples.com/pyspark/pyspark-arraytype-column-with-examples/) for arrays, and [MapType](https://sparkbyexamples.com/pyspark/pyspark-maptype-dict-examples/) for key-value pairs which we will discuss in detail in later sections.

The below example demonstrates a very simple example of how to create a StructType & StructField on DataFrame and it’s usage with sample data to support it.

import pyspark

from pyspark.sql import SparkSession

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

spark = SparkSession.builder.master("local[1]") \

.appName('SparkByExamples.com') \

.getOrCreate()

data = [("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=data,schema=schema)

df.printSchema()

df.show(truncate=False)

By running the above snippet, it displays below outputs.

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 |

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

## 4. Defining Nested StructType object struct

While working on DataFrame we often need to work with the nested struct column and this can be defined using StructType.

On the below example column “name” data type is StructType which is nested.

structureData = [

(("James","","Smith"),"36636","M",3100),

(("Michael","Rose",""),"40288","M",4300),

(("Robert","","Williams"),"42114","M",1400),

(("Maria","Anne","Jones"),"39192","F",5500),

(("Jen","Mary","Brown"),"","F",-1)

]

structureSchema = StructType([

StructField('name', 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)

])

df2 = spark.createDataFrame(data=structureData,schema=structureSchema)

df2.printSchema()

df2.show(truncate=False)

Outputs below schema and the DataFrame

root

|-- name: struct (nullable = true)

| |-- firstname: string (nullable = true)

| |-- middlename: string (nullable = true)

| |-- lastname: string (nullable = true)

|-- id: string (nullable = true)

|-- gender: string (nullable = true)

|-- salary: integer (nullable = true)

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

|name |id |gender|salary|

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

|[James, , Smith] |36636|M |3100 |

|[Michael, Rose, ] |40288|M |4300 |

|[Robert, , Williams]|42114|M |1400 |

|[Maria, Anne, Jones]|39192|F |5500 |

|[Jen, Mary, Brown] | |F |-1 |

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

## 5. Adding & Changing struct of the DataFrame

Using [PySpark SQL function](https://sparkbyexamples.com/spark/spark-sql-functions-understanding/) struct(), we can change the struct of the existing DataFrame and add a new StructType to it. The below example demonstrates how to copy the columns from one structure to another and adding a new column. [PySpark Column Class](https://sparkbyexamples.com/pyspark/pyspark-column-functions/) also provides some functions to work with the StructType column.

from pyspark.sql.functions import col,struct,when

updatedDF = df2.withColumn("OtherInfo",

struct(col("id").alias("identifier"),

col("gender").alias("gender"),

col("salary").alias("salary"),

when(col("salary").cast(IntegerType()) < 2000,"Low")

.when(col("salary").cast(IntegerType()) < 4000,"Medium")

.otherwise("High").alias("Salary\_Grade")

)).drop("id","gender","salary")

updatedDF.printSchema()

updatedDF.show(truncate=False)

Here, it copies “gender“, “salary” and “id” to the new struct “otherInfo” and add’s a new column “Salary\_Grade“.

root

|-- name: struct (nullable = true)

| |-- firstname: string (nullable = true)

| |-- middlename: string (nullable = true)

| |-- lastname: string (nullable = true)

|-- OtherInfo: struct (nullable = false)

| |-- identifier: string (nullable = true)

| |-- gender: string (nullable = true)

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

| |-- Salary\_Grade: string (nullable = false)

## 6. Using SQL ArrayType and MapType

SQL StructType also supports [ArrayType](https://sparkbyexamples.com/pyspark/pyspark-arraytype-column-with-examples/) and [MapType](https://sparkbyexamples.com/pyspark/pyspark-maptype-dict-examples/) to define the DataFrame columns for array and map collections respectively. On the below example, column hobbies defined as ArrayType(StringType) and properties defined as MapType(StringType,StringType) meaning both key and value as String.

arrayStructureSchema = StructType([

StructField('name', StructType([

StructField('firstname', StringType(), True),

StructField('middlename', StringType(), True),

StructField('lastname', StringType(), True)

])),

StructField('hobbies', ArrayType(StringType()), True),

StructField('properties', MapType(StringType(),StringType()), True)

])

Outputs the below schema. Note that field Hobbies is array type and properties is map type.

root

|-- name: struct (nullable = true)

| |-- firstname: string (nullable = true)

| |-- middlename: string (nullable = true)

| |-- lastname: string (nullable = true)

|-- hobbies: array (nullable = true)

| |-- element: string (containsNull = true)

|-- properties: map (nullable = true)

| |-- key: string

| |-- value: string (valueContainsNull = true)

## 7. Creating StructType object struct from JSON file

If you have too many columns and the structure of the DataFrame changes now and then, it’s a good practice to load the SQL StructType schema from JSON file. You can get the schema by using df2.schema.json() , store this in a file and will use it to create a the schema from this file.

print(df2.schema.json())

{

"type" : "struct",

"fields" : [ {

"name" : "name",

"type" : {

"type" : "struct",

"fields" : [ {

"name" : "firstname",

"type" : "string",

"nullable" : true,

"metadata" : { }

}, {

"name" : "middlename",

"type" : "string",

"nullable" : true,

"metadata" : { }

}, {

"name" : "lastname",

"type" : "string",

"nullable" : true,

"metadata" : { }

} ]

},

"nullable" : true,

"metadata" : { }

}, {

"name" : "dob",

"type" : "string",

"nullable" : true,

"metadata" : { }

}, {

"name" : "gender",

"type" : "string",

"nullable" : true,

"metadata" : { }

}, {

"name" : "salary",

"type" : "integer",

"nullable" : true,

"metadata" : { }

} ]

}

Alternatively, you could also use df.schema.simpleString(), this will return an relatively simpler schema format.

Now let’s load the json file and use it to create a DataFrame.

import json

schemaFromJson = StructType.fromJson(json.loads(schema.json))

df3 = spark.createDataFrame(

spark.sparkContext.parallelize(structureData),schemaFromJson)

df3.printSchema()

This prints the same output as the previous section. You can also, have a name, type, and flag for nullable in a comma-separated file and we can use these to create a StructType programmatically, I will leave this to you to explore.

## 8. Creating StructType object struct from DDL String

Like loading structure from JSON string, we can also create it from DLL ( by using fromDDL() static function on SQL StructType class StructType.fromDDL). You can also generate DDL from a schema using toDDL(). printTreeString() on struct object prints the schema similar to printSchemafunction returns.

ddlSchemaStr = "`fullName` STRUCT<`first`: STRING, `last`: STRING,

`middle`: STRING>,`age` INT,`gender` STRING"

ddlSchema = StructType.fromDDL(ddlSchemaStr)

ddlSchema.printTreeString()

## 9. Checking if a Column Exists in a DataFrame

If you want to perform some checks on metadata of the DataFrame, for example, if a column or field exists in a DataFrame or data type of column; we can easily do this using several functions on SQL StructType and StructField.

print(df.schema.fieldNames.contains("firstname"))

print(df.schema.contains(StructField("firstname",StringType,true)))

This example returns “true” for both scenarios. And for the second one if you have IntegerType instead of StringType it returns false as the datatype for first name column is String, as it checks every property in a field. Similarly, you can also check if two schemas are equal and more.

## 10. Complete Example of PySpark StructType & StructField

import pyspark

from pyspark.sql import SparkSession

from pyspark.sql.types import StructType,StructField, StringType, IntegerType,ArrayType,MapType

from pyspark.sql.functions import col,struct,when

spark = SparkSession.builder.master("local[1]") \

.appName('SparkByExamples.com') \

.getOrCreate()

data = [("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=data,schema=schema)

df.printSchema()

df.show(truncate=False)

structureData = [

(("James","","Smith"),"36636","M",3100),

(("Michael","Rose",""),"40288","M",4300),

(("Robert","","Williams"),"42114","M",1400),

(("Maria","Anne","Jones"),"39192","F",5500),

(("Jen","Mary","Brown"),"","F",-1)

]

structureSchema = StructType([

StructField('name', 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)

])

df2 = spark.createDataFrame(data=structureData,schema=structureSchema)

df2.printSchema()

df2.show(truncate=False)

updatedDF = df2.withColumn("OtherInfo",

struct(col("id").alias("identifier"),

col("gender").alias("gender"),

col("salary").alias("salary"),

when(col("salary").cast(IntegerType()) < 2000,"Low")

.when(col("salary").cast(IntegerType()) < 4000,"Medium")

.otherwise("High").alias("Salary\_Grade")

)).drop("id","gender","salary")

updatedDF.printSchema()

updatedDF.show(truncate=False)

""" Array & Map"""

arrayStructureSchema = StructType([

StructField('name', StructType([

StructField('firstname', StringType(), True),

StructField('middlename', StringType(), True),

StructField('lastname', StringType(), True)

])),

StructField('hobbies', ArrayType(StringType()), True),

StructField('properties', MapType(StringType(),StringType()), True)

])

The complete example explained here is available also available at [GitHub](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-structtype.py) project.

#### Conclusion:

In this article, you have learned the usage of SQL StructType, StructField, and how to change the structure of the Pyspark DataFrame at runtime, converting case class to the schema and using ArrayType, MapType.

# Row using on DataFrame and RDD

In PySpark Row class is available by importing pyspark.sql.Row which is represented as a record/row in DataFrame, one can create a Row object by using named arguments, or create a custom Row like class. In this article I will explain how to use Row class on RDD, DataFrame and its functions.

Before we start using it on RDD & DataFrame, let’s understand some basics of Row class.

**Related Article:** [PySpark Column Class Usage & Functions with Examples](https://sparkbyexamples.com/pyspark/pyspark-column-functions/)

**Key Points of Row Class:**

* Earlier to Spark 3.0, when used Row class with named arguments, the fields are sorted by name.
* Since 3.0, Rows created from named arguments are not sorted alphabetically instead they will be ordered in the position entered.
* To enable sorting by names, set the environment variable PYSPARK\_ROW\_FIELD\_SORTING\_ENABLED to true.
* Row class provides a way to create a struct-type column as well.

## 1. Create a Row Object

Row class extends the tuple hence it takes variable number of arguments, Row() is used to create the row object. Once the row object created, we can retrieve the data from Row using index similar to tuple.

from pyspark.sql import Row

row=Row("James",40)

print(row[0] +","+str(row[1]))

This outputs James,40. Alternatively you can also write with named arguments. Benefits with the named argument is you can access with field name row.name. Below example print “Alice”.

row=Row(name="Alice", age=11)

print(row.name)

## 2. Create Custom Class from Row

We can also create a Row like class, for example “Person” and use it similar to Row object. This would be helpful when you wanted to create real time object and refer it’s properties. On below example, we have created a Person class and used similar to Row.

Person = Row("name", "age")

p1=Person("James", 40)

p2=Person("Alice", 35)

print(p1.name +","+p2.name)

This outputs James,Alice

## 3. Using Row class on PySpark RDD

We can use Row class on PySpark RDD. When you use Row to create an RDD, after collecting the data you will get the result back in Row.

from pyspark.sql import SparkSession, Row

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

data = [Row(name="James,,Smith",lang=["Java","Scala","C++"],state="CA"),

Row(name="Michael,Rose,",lang=["Spark","Java","C++"],state="NJ"),

Row(name="Robert,,Williams",lang=["CSharp","VB"],state="NV")]

rdd=spark.sparkContext.parallelize(data)

print(rdd.collect())

This yields below output.

[Row(name='James,,Smith', lang=['Java', 'Scala', 'C++'], state='CA'), Row(name='Michael,Rose,', lang=['Spark', 'Java', 'C++'], state='NJ'), Row(name='Robert,,Williams', lang=['CSharp', 'VB'], state='NV')]

Now, let’s collect the data and access the data using its properties.

collData=rdd.collect()

for row in collData:

print(row.name + "," +str(row.lang))

This yields below output.

James,,Smith,['Java', 'Scala', 'C++']

Michael,Rose,,['Spark', 'Java', 'C++']

Robert,,Williams,['CSharp', 'VB']

Alternatively, you can also do by creating a Row like class “Person”

Person=Row("name","lang","state")

data = [Person("James,,Smith",["Java","Scala","C++"],"CA"),

Person("Michael,Rose,",["Spark","Java","C++"],"NJ"),

Person("Robert,,Williams",["CSharp","VB"],"NV")]

## 4. Using Row class on PySpark DataFrame

Similarly, Row class also can be used with PySpark DataFrame, By default data in DataFrame represent as Row. To demonstrate, I will use the same data that was created for RDD.

Note that Row on DataFrame is not allowed to omit a named argument to represent that the value is None or missing. This should be explicitly set to None in this case.

df=spark.createDataFrame(data)

df.printSchema()

df.show()

This yields below output. Note that DataFrame able to take the column names from Row object.

root

|-- name: string (nullable = true)

|-- lang: array (nullable = true)

| |-- element: string (containsNull = true)

|-- state: string (nullable = true)

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

| name| lang|state|

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

| James,,Smith|[Java, Scala, C++]| CA|

| Michael,Rose,|[Spark, Java, C++]| NJ|

|Robert,,Williams| [CSharp, VB]| NV|

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

You can also change the column names by using toDF() function

columns = ["name","languagesAtSchool","currentState"]

df=spark.createDataFrame(data).toDF(\*columns)

df.printSchema()

This yields below output, note the column name “languagesAtSchool” from the previous example.

root

|-- name: string (nullable = true)

|-- languagesAtSchool: array (nullable = true)

| |-- element: string (containsNull = true)

|-- currentState: string (nullable = true)

## 5. Create Nested Struct Using Row Class

The below example provides a way to create a struct type using the Row class. Alternatively, you can also create struct type using By [Providing Schema using PySpark StructType & StructFields](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/)

#Create DataFrame with struct using Row class

from pyspark.sql import Row

data=[Row(name="James",prop=Row(hair="black",eye="blue")),

Row(name="Ann",prop=Row(hair="grey",eye="black"))]

df=spark.createDataFrame(data)

df.printSchema()

Yields below schema

root

|-- name: string (nullable = true)

|-- prop: struct (nullable = true)

| |-- hair: string (nullable = true)

| |-- eye: string (nullable = true)

## 6. Complete Example of PySpark Row usage on RDD & DataFrame

Below is complete example for reference.

from pyspark.sql import SparkSession, Row

row=Row("James",40)

print(row[0] +","+str(row[1]))

row2=Row(name="Alice", age=11)

print(row2.name)

Person = Row("name", "age")

p1=Person("James", 40)

p2=Person("Alice", 35)

print(p1.name +","+p2.name)

#PySpark Example

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

data = [Row(name="James,,Smith",lang=["Java","Scala","C++"],state="CA"),

Row(name="Michael,Rose,",lang=["Spark","Java","C++"],state="NJ"),

Row(name="Robert,,Williams",lang=["CSharp","VB"],state="NV")]

#RDD Example 1

rdd=spark.sparkContext.parallelize(data)

collData=rdd.collect()

print(collData)

for row in collData:

print(row.name + "," +str(row.lang))

# RDD Example 2

Person=Row("name","lang","state")

data = [Person("James,,Smith",["Java","Scala","C++"],"CA"),

Person("Michael,Rose,",["Spark","Java","C++"],"NJ"),

Person("Robert,,Williams",["CSharp","VB"],"NV")]

rdd=spark.sparkContext.parallelize(data)

collData=rdd.collect()

print(collData)

for person in collData:

print(person.name + "," +str(person.lang))

#DataFrame Example 1

columns = ["name","languagesAtSchool","currentState"]

df=spark.createDataFrame(data)

df.printSchema()

df.show()

collData=df.collect()

print(collData)

for row in collData:

print(row.name + "," +str(row.lang))

#DataFrame Example 2

columns = ["name","languagesAtSchool","currentState"]

df=spark.createDataFrame(data).toDF(\*columns)

df.printSchema()

#### Conclusion

In this PySpark Row article you have learned how to use Row class with named argument and defining realtime class and using it on DataFrame & RDD. Hope you like this

#### Reference

* <https://spark.apache.org/docs/latest/api/python/pyspark.sql.html>

# Column Class | Operators & Functions

pyspark.sql.Column class provides several functions to work with DataFrame to manipulate the Column values, evaluate the boolean expression to filter rows, retrieve a value or part of a value from a DataFrame column, and to work with list, map & struct columns.

In this article, I will cover how to create Column object, access them to perform operations, and finally most used PySpark Column Functions with Examples.

**Related Article:** [PySpark Row Class with Examples](https://sparkbyexamples.com/pyspark/pyspark-row-using-rdd-dataframe/)

**Key Points:**

* PySpark Column class represents a single Column in a DataFrame.
* It provides functions that are most used to manipulate DataFrame Columns & Rows.
* Some of these Column functions evaluate a Boolean expression that can be used with filter() transformation to [filter the DataFrame Rows](https://sparkbyexamples.com/pyspark/pyspark-where-filter/).
* Provides functions to get a value from a list column by index, map value by key & index, and finally struct nested column.
* PySpark also provides additional functions [pyspark.sql.functions](https://spark.apache.org/docs/latest/api/python/_modules/pyspark/sql/functions.html) that take Column object and return a Column type.

**Note:** Most of the [pyspark.sql.functions](https://spark.apache.org/docs/latest/api/python/_modules/pyspark/sql/functions.html) return Column type hence it is very important to know the operation you can perform with Column type.

## 1. Create Column Class Object

One of the simplest ways to create a Column class object is by using [PySpark lit() SQL function](https://sparkbyexamples.com/pyspark/pyspark-lit-add-literal-constant/), this takes a literal value and returns a Column object.

from pyspark.sql.functions import lit

colObj = lit("sparkbyexamples.com")

You can also access the Column from DataFrame by multiple ways.

data=[("James",23),("Ann",40)]

df=spark.createDataFrame(data).toDF("name.fname","gender")

df.printSchema()

#root

# |-- name.fname: string (nullable = true)

# |-- gender: long (nullable = true)

# Using DataFrame object (df)

df.select(df.gender).show()

df.select(df["gender"]).show()

#Accessing column name with dot (with backticks)

df.select(df["`name.fname`"]).show()

#Using SQL col() function

from pyspark.sql.functions import col

df.select(col("gender")).show()

#Accessing column name with dot (with backticks)

df.select(col("`name.fname`")).show()

Below example demonstrates accessing struct type columns. Here I have use [PySpark Row class](https://sparkbyexamples.com/pyspark/pyspark-row-using-rdd-dataframe/) to create a struct type. Alternatively you can also create it by using [PySpark StructType & StructField classes](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/)

Create DataFrame with struct using Row class

from pyspark.sql import Row

data=[Row(name="James",prop=Row(hair="black",eye="blue")),

Row(name="Ann",prop=Row(hair="grey",eye="black"))]

df=spark.createDataFrame(data)

df.printSchema()

#root

# |-- name: string (nullable = true)

# |-- prop: struct (nullable = true)

# | |-- hair: string (nullable = true)

# | |-- eye: string (nullable = true)

#Access struct column

df.select(df.prop.hair).show()

df.select(df["prop.hair"]).show()

df.select(col("prop.hair")).show()

#Access all columns from struct

df.select(col("prop.\*")).show()

## 2. PySpark Column Operators

PySpark column also provides a way to do arithmetic operations on columns using operators.

data=[(100,2,1),(200,3,4),(300,4,4)]

df=spark.createDataFrame(data).toDF("col1","col2","col3")

#Arthmetic operations

df.select(df.col1 + df.col2).show()

df.select(df.col1 - df.col2).show()

df.select(df.col1 \* df.col2).show()

df.select(df.col1 / df.col2).show()

df.select(df.col1 % df.col2).show()

df.select(df.col2 > df.col3).show()

df.select(df.col2 < df.col3).show()

df.select(df.col2 == df.col3).show()

## 3. PySpark Column Functions

Let’s see some of the most used Column Functions, on below table, I have grouped related functions together to make it easy, click on the link for examples.

| COLUMN FUNCTION | FUNCTION DESCRIPTION |
| --- | --- |
| alias(\*alias, \*\*kwargs) name(\*alias, \*\*kwargs) | Provides alias to the column or expressions name() returns same as alias(). |
| asc() asc\_nulls\_first() asc\_nulls\_last() | Returns ascending order of the column. asc\_nulls\_first() Returns null values first then non-null values. asc\_nulls\_last() – Returns null values after non-null values. |
| astype(dataType) cast(dataType) | Used to cast the data type to another type. astype() returns same as cast(). |
| between(lowerBound, upperBound) | Checks if the columns values are between lower and upper bound. Returns boolean value. |
| bitwiseAND(other) bitwiseOR(other) bitwiseXOR(other) | Compute bitwise AND, OR & XOR of this expression with another expression respectively. |
| contains(other) | Check if String contains in another string. |
| desc() desc\_nulls\_first() desc\_nulls\_last() | Returns descending order of the column. desc\_nulls\_first() -null values appear before non-null values. desc\_nulls\_last() – null values appear after non-null values. |
| startswith(other) endswith(other) | String starts with. Returns boolean expression String ends with. Returns boolean expression |
| eqNullSafe(other) | Equality test that is safe for null values. |
| getField(name) | Returns a field by name in a StructField and by key in Map. |
| getItem(key) | Returns a values from Map/Key at the provided position. |
| isNotNull() isNull() | isNotNull() – Returns True if the current expression is NOT null. isNull() – Returns True if the current expression is null. |
| isin(\*cols) | A boolean expression that is evaluated to true if the value of this expression is contained by the evaluated values of the arguments. |
| like(other) rlike(other) | Similar to SQL like expression. Similar to SQL RLIKE expression (LIKE with Regex). |
| over(window) | Used with window column |
| substr(startPos, length) | Return a Column which is a substring of the column. |
| when(condition, value) otherwise(value) | Similar to SQL CASE WHEN, Executes a list of conditions and returns one of multiple possible result expressions. |
| dropFields(\*fieldNames) | Used to drops fields in StructType by name. |
| withField(fieldName, col) | An expression that adds/replaces a field in StructType by name. |

## 4. PySpark Column Functions Examples

Let’s create a simple DataFrame to work with PySpark SQL Column examples. For most of the examples below, I will be referring DataFrame object name (df.) to get the column.

data=[("James","Bond","100",None),

("Ann","Varsa","200",'F'),

("Tom Cruise","XXX","400",''),

("Tom Brand",None,"400",'M')]

columns=["fname","lname","id","gender"]

df=spark.createDataFrame(data,columns)

## 4.1 alias() – Set’s name to Column

On below example df.fname refers to Column object and alias() is a function of the Column to give alternate name. Here, fname column has been changed to first\_name & lname to last\_name.

On second example I have use [PySpark expr() function to concatenate columns](https://sparkbyexamples.com/pyspark/pyspark-sql-expr-expression-function/) and named column as fullName.

#alias

from pyspark.sql.functions import expr

df.select(df.fname.alias("first\_name"), \

df.lname.alias("last\_name")

).show()

#Another example

df.select(expr(" fname ||','|| lname").alias("fullName") \

).show()

### 4.2 asc() & desc() – Sort the DataFrame columns by Ascending or Descending order.

#asc, desc to sort ascending and descending order repsectively.

df.sort(df.fname.asc()).show()

df.sort(df.fname.desc()).show()

## 4.3 cast() & astype() – Used to convert the data Type.

#cast

df.select(df.fname,df.id.cast("int")).printSchema()

### 4.4 between() – Returns a Boolean expression when a column values in between lower and upper bound.

#between

df.filter(df.id.between(100,300)).show()

### 4.5 contains() – Checks if a DataFrame column value contains a a value specified in this function.

#contains

df.filter(df.fname.contains("Cruise")).show()

### 4.6 startswith() & endswith() – Checks if the value of the DataFrame Column starts and ends with a String respectively.

#startswith, endswith()

df.filter(df.fname.startswith("T")).show()

df.filter(df.fname.endswith("Cruise")).show()

### 4.7 eqNullSafe() –

### 4.8 isNull & isNotNull() – Checks if the DataFrame column has NULL or non NULL values.

Refer to

#isNull & isNotNull

df.filter(df.lname.isNull()).show()

df.filter(df.lname.isNotNull()).show()

### 4.9 like() & rlike() – Similar to SQL LIKE expression

#like , rlike

df.select(df.fname,df.lname,df.id) \

.filter(df.fname.like("%om"))

### 4.10 substr() – Returns a Column after getting sub string from the Column

df.select(df.fname.substr(1,2).alias("substr")).show()

### 4.11 when() & otherwise() – It is similar to SQL Case When, executes sequence of expressions until it matches the condition and returns a value when match.

#when & otherwise

from pyspark.sql.functions import when

df.select(df.fname,df.lname,when(df.gender=="M","Male") \

.when(df.gender=="F","Female") \

.when(df.gender==None ,"") \

.otherwise(df.gender).alias("new\_gender") \

).show()

### 4.12 isin() – Check if value presents in a List.

#isin

li=["100","200"]

df.select(df.fname,df.lname,df.id) \

.filter(df.id.isin(li)) \

.show()

### 4.13 getField() – To get the value by key from MapType column and by stuct child name from StructType column

Rest of the below functions operates on List, Map & Struct data structures hence to demonstrate these I will use another DataFrame with list, map and struct columns. For more explanation how to use Arrays refer to [PySpark ArrayType Column on DataFrame Examples](https://sparkbyexamples.com/pyspark/pyspark-arraytype-column-with-examples/) & for map refer to [PySpark MapType Examples](https://sparkbyexamples.com/pyspark/pyspark-maptype-dict-examples/)

#Create DataFrame with struct, array & map

from pyspark.sql.types import StructType,StructField,StringType,ArrayType,MapType

data=[(("James","Bond"),["Java","C#"],{'hair':'black','eye':'brown'}),

(("Ann","Varsa"),[".NET","Python"],{'hair':'brown','eye':'black'}),

(("Tom Cruise",""),["Python","Scala"],{'hair':'red','eye':'grey'}),

(("Tom Brand",None),["Perl","Ruby"],{'hair':'black','eye':'blue'})]

schema = StructType([

StructField('name', StructType([

StructField('fname', StringType(), True),

StructField('lname', StringType(), True)])),

StructField('languages', ArrayType(StringType()),True),

StructField('properties', MapType(StringType(),StringType()),True)

])

df=spark.createDataFrame(data,schema)

df.printSchema()

#Display's to console

root

|-- name: struct (nullable = true)

| |-- fname: string (nullable = true)

| |-- lname: string (nullable = true)

|-- languages: array (nullable = true)

| |-- element: string (containsNull = true)

|-- properties: map (nullable = true)

| |-- key: string

| |-- value: string (valueContainsNull = true)

getField Example

#getField from MapType

df.select(df.properties.getField("hair")).show()

#getField from Struct

df.select(df.name.getField("fname")).show()

### 4.14 getItem() – To get the value by index from MapType or ArrayTupe & ny key for MapType column.

#getItem() used with ArrayType

df.select(df.languages.getItem(1)).show()

#getItem() used with MapType

df.select(df.properties.getItem("hair")).show()

### 4.15 dropFields –

# TO-DO, getting runtime error

### 4.16 withField() –

# TO-DO getting runtime error

### 4.17 over() – Used with Window Functions

TO-DO

# PySpark Select Columns From DataFrame

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](https://sparkbyexamples.com/pyspark/select-columns-from-pyspark-dataframe/#select-multiple-columns)
* [Select All Columns From List](https://sparkbyexamples.com/pyspark/select-columns-from-pyspark-dataframe/#select-all-columns-from-list)
* [Select Columns By Index](https://sparkbyexamples.com/pyspark/select-columns-from-pyspark-dataframe/#select-columns-by-index)
* [Select a Nested Column](https://sparkbyexamples.com/pyspark/select-columns-from-pyspark-dataframe/#select-nested-column)
* [Other Ways to Select Columns](https://sparkbyexamples.com/pyspark/select-columns-from-pyspark-dataframe/#ways-to-select)

First, let’s [create a Dataframe](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/).

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)

## 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()

## 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()

## 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)

## 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](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/) 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

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 |

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

Now, let’s select struct column.

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

This returns struct column name as is.

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

|name |

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

|[James,, Smith] |

|[Anna, Rose, ] |

|[Julia, , Williams] |

|[Maria, Anne, Jones] |

|[Jen, Mary, Brown] |

|[Mike, Mary, Williams]|

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

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)

This outputs firstname and lastname from the name struct column.

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

|firstname|lastname|

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

|James |Smith |

|Anna | |

|Julia |Williams|

|Maria |Jones |

|Jen |Brown |

|Mike |Williams|

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

In order to get all columns from struct column.

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

This yields below output.

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

|firstname|middlename|lastname|

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

|James |null |Smith |

|Anna |Rose | |

|Julia | |Williams|

|Maria |Anne |Jones |

|Jen |Mary |Brown |

|Mike |Mary |Williams|

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

## 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)

This example is also available at [PySpark github project](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-select-columns.py).

## 6. Conclusion

In this article, you have learned select() is a transformation function of the DataFrame and is used to select single, multiple columns, select all columns from the list, select by index, and finally select nested struct columns, you have also learned how to select nested elements from the DataFrame.

## Related Articles

* [How to Replace Column Values in PySpark DataFrame](https://sparkbyexamples.com/pyspark/pyspark-replace-column-values/)
* [How to Retrieve DataType & Column Names of PySpark DataFrame](https://sparkbyexamples.com/pyspark/pyspark-find-datatype-column-names-of-dataframe/)

# PySpark Collect() – Retrieve data from DataFrame

PySpark RDD/DataFrame collect() is an action operation that is used to retrieve all the elements of the dataset (from all nodes) to the driver node. We should use the collect() on smaller dataset usually after [filter()](https://sparkbyexamples.com/pyspark/pyspark-where-filter/), [group()](https://sparkbyexamples.com/pyspark/pyspark-groupby-explained-with-example/) e.t.c. Retrieving larger datasets results in OutOfMemory error.

In this PySpark article, I will explain the usage of collect() with DataFrame example, when to avoid it, and the difference between collect() and select().

**Related Articles:**

* [How to Iterate PySpark DataFrame through Loop](https://sparkbyexamples.com/pyspark/pyspark-loop-iterate-through-rows-in-dataframe/)
* [How to Convert PySpark DataFrame Column to Python List](https://sparkbyexamples.com/pyspark/convert-pyspark-dataframe-column-to-python-list/)

In order to explain with example, first, let’s [create a DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/).

import pyspark

from pyspark.sql import SparkSession

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

dept = [("Finance",10), \

("Marketing",20), \

("Sales",30), \

("IT",40) \

]

deptColumns = ["dept\_name","dept\_id"]

deptDF = spark.createDataFrame(data=dept, schema = deptColumns)

deptDF.show(truncate=False)

[show() function on DataFrame prints the result of DataFrame in a table format](https://sparkbyexamples.com/pyspark/pyspark-show-display-dataframe-contents-in-table/). By default, it shows only 20 rows. The above snippet returns the data in a table.

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

|dept\_name|dept\_id|

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

|Finance |10 |

|Marketing|20 |

|Sales |30 |

|IT |40 |

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

Now, let’s use the collect() to retrieve the data.

dataCollect = deptDF.collect()

print(dataCollect)

deptDF.collect() retrieves all elements in a DataFrame as an Array of [Row type](https://sparkbyexamples.com/pyspark/pyspark-row-using-rdd-dataframe/) to the driver node. printing a resultant array yields the below output.

[Row(dept\_name='Finance', dept\_id=10),

Row(dept\_name='Marketing', dept\_id=20),

Row(dept\_name='Sales', dept\_id=30),

Row(dept\_name='IT', dept\_id=40)]

Note that collect() is an action hence it does not return a DataFrame instead, it returns data in an Array to the driver. Once the data is in an array, you can use python for loop to process it further.

for row in dataCollect:

print(row['dept\_name'] + "," +str(row['dept\_id']))

If you wanted to get first row and first column from a DataFrame.

#Returns value of First Row, First Column which is "Finance"

deptDF.collect()[0][0]

Let’s understand what’s happening on above statement.

* deptDF.collect() returns Array of Row type.
* deptDF.collect()[0] returns the first element in an array (1st row).
* deptDF.collect[0][0] returns the value of the first row & first column.

In case you want to just return certain elements of a DataFrame, you should call [PySpark select() transformation](https://sparkbyexamples.com/pyspark/select-columns-from-pyspark-dataframe/) first.

dataCollect = deptDF.select("dept\_name").collect()

## When to avoid Collect()

Usually, collect() is used to retrieve the action output when you have very small result set and calling collect() on an RDD/DataFrame with a bigger result set causes out of memory as it returns the entire dataset (from all workers) to the driver hence we should avoid calling collect() on a larger dataset.

## collect () vs select ()

select() is a transformation that returns a new DataFrame and holds the columns that are selected whereas collect() is an action that returns the entire data set in an Array to the driver.

## Complete Example of PySpark collect()

Below is complete PySpark example of using collect() on DataFrame, similarly you can also create a program using collect() with RDD.

import pyspark

from pyspark.sql import SparkSession

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

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)

dataCollect = deptDF.collect()

print(dataCollect)

dataCollect2 = deptDF.select("dept\_name").collect()

print(dataCollect2)

for row in dataCollect:

print(row['dept\_name'] + "," +str(row['dept\_id']))

This example is also available at [PySpark Github](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-collect.py) project.

## Conclusion

In this PySpark article, you have learned the collect() function of the RDD/DataFrame is an action operation that returns all elements of the DataFrame to spark driver program and also learned it’s not a good practice to use it on the bigger dataset.

# PySpark withColumn() Usage with Examples

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.

First, let’s[create a DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/) 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)

## 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](https://sparkbyexamples.com/pyspark/pyspark-cast-column-type/), you would also need to use cast() function along with withColumn(). The below statement [changes the datatype from String to Integer](https://sparkbyexamples.com/spark/spark-cast-string-type-to-integer-type-int/) for the salary column.

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

## 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](https://sparkbyexamples.com/pyspark/pyspark-update-a-column-with-value/).

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

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](https://sparkbyexamples.com/pyspark/pyspark-add-new-column-to-dataframe/).

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

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](https://sparkbyexamples.com/pyspark/pyspark-lit-add-literal-constant/). 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()

## 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()](https://sparkbyexamples.com/pyspark/pyspark-rename-dataframe-column/) function on DataFrame.

df.withColumnRenamed("gender","sex") \

.show(truncate=False)

## 6. Drop Column From PySpark DataFrame

Use “drop” function to [drop a specific column from the DataFrame](https://sparkbyexamples.com/pyspark/pyspark-drop-column-from-dataframe/).

df.drop("salary") \

.show()

**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)

The complete code can be downloaded from [PySpark withColumn GitHub project](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-withcolumn.py)

# PySpark withColumnRenamed to Rename Column on DataFrame

Use PySpark withColumnRenamed() to rename a DataFrame column, we often need to rename one column or multiple (or all) columns on PySpark DataFrame, you can do this in several ways. When columns are nested it becomes complicated.

Since DataFrame’s are an immutable collection, you can’t rename or update a column instead when using withColumnRenamed() it creates a new DataFrame with updated column names, In this PySpark article, I will cover different ways to rename columns with several use cases like rename nested column, all columns, selected multiple columns with Python/PySpark examples.

First, let’s create our data set to work with.

dataDF = [(('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)

]

Our base schema with nested structure.

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

schema = StructType([

StructField('name', StructType([

StructField('firstname', StringType(), True),

StructField('middlename', StringType(), True),

StructField('lastname', StringType(), True)

])),

StructField('dob', StringType(), True),

StructField('gender', StringType(), True),

StructField('gender', IntegerType(), True)

])

Let’s create the DataFrame by using parallelize and provide the above schema.

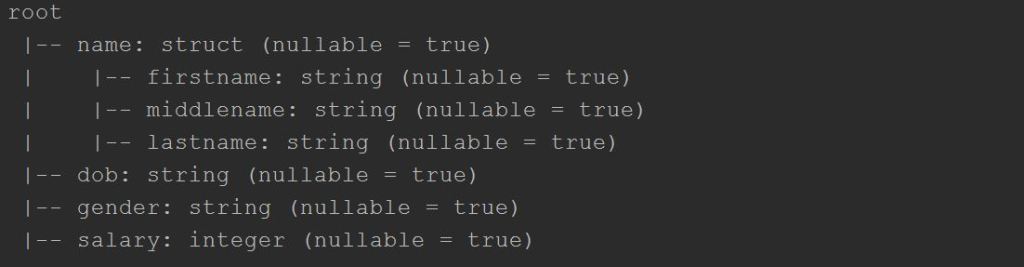
from pyspark.sql import SparkSession

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

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

df.printSchema()

Below is our schema structure. I am not printing data here as it is not necessary for our examples. This schema has a nested structure.



## 1. PySpark withColumnRenamed – To rename DataFrame column name

PySpark has a withColumnRenamed() function on DataFrame to change a column name. This is the most straight forward approach; this function takes two parameters; the first is your existing column name and the second is the new column name you wish for.

**PySpark withColumnRenamed**() **Syntax:**

withColumnRenamed(existingName, newNam)

existingName – The existing column name you want to change

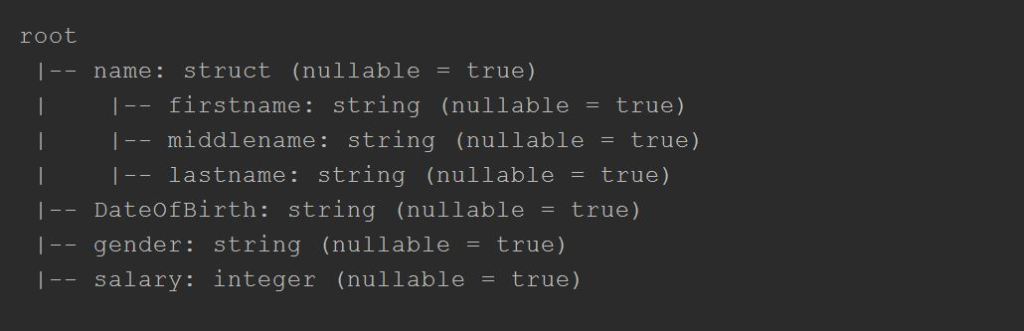
newName – New name of the column

Returns a new DataFrame with a column renamed.

**Example**

df.withColumnRenamed("dob","DateOfBirth").printSchema()

The above statement changes column “dob” to “DateOfBirth” on PySpark DataFrame. Note that withColumnRenamed function returns a new DataFrame and doesn’t modify the current DataFrame.



## 2. PySpark withColumnRenamed – To rename multiple columns

To change multiple column names, we should chain withColumnRenamed functions as shown below. You can also store all columns to rename in a list and loop through to rename all columns, I will leave this to you to explore.

df2 = df.withColumnRenamed("dob","DateOfBirth") \

.withColumnRenamed("salary","salary\_amount")

df2.printSchema()

This creates a new DataFrame “df2” after renaming dob and salary columns.

## 3. Using PySpark StructType – To rename a nested column in Dataframe

Changing a column name on nested data is not straight forward and we can do this by creating a new schema with new [DataFrame columns using StructType](https://sparkbyexamples.com/spark/spark-sql-structtype-on-dataframe/) and use it using cast function as shown below.

schema2 = StructType([

StructField("fname",StringType()),

StructField("middlename",StringType()),

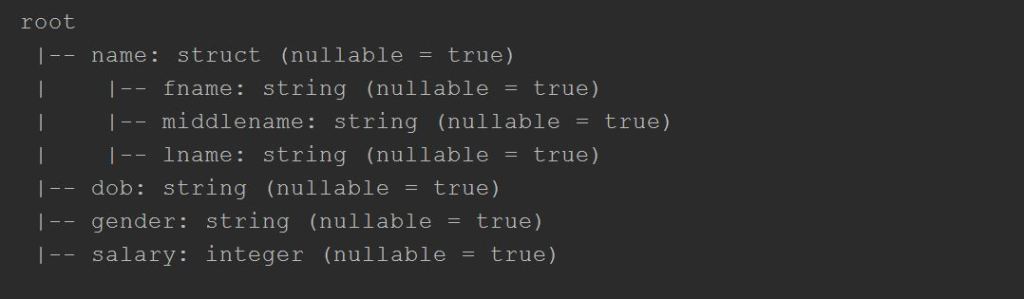
StructField("lname",StringType())])

df.select(col("name").cast(schema2), \

col("dob"), col("gender"),col("salary")) \

.printSchema()

This statement renames firstname to fname and lastname to lname within name structure.



## 4. Using Select – To rename nested elements.

Let’s see another way to change nested columns by transposing the structure to flat.

from pyspark.sql.functions import \*

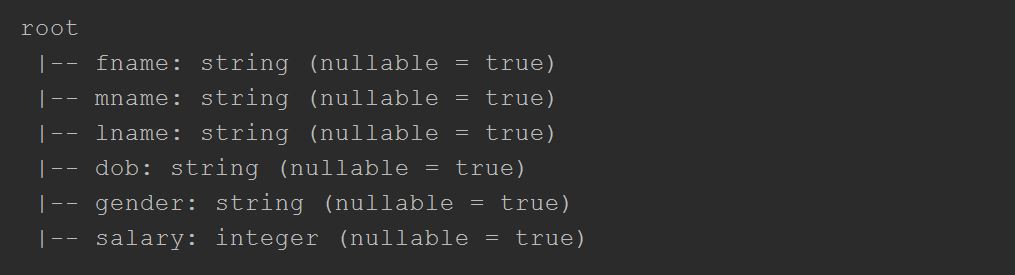
df.select(col("name.firstname").alias("fname"), \

col("name.middlename").alias("mname"), \

col("name.lastname").alias("lname"), \

col("dob"),col("gender"),col("salary")) \

.printSchema()



## 5. Using PySpark DataFrame withColumn – To rename nested columns

When you have nested columns on PySpark DatFrame and if you want to rename it, use withColumn on a data frame object to create a new column from an existing and we will need to drop the existing column. Below example creates a “fname” column from “name.firstname” and drops the “name” column

from pyspark.sql.functions import \*

df4 = df.withColumn("fname",col("name.firstname")) \

.withColumn("mname",col("name.middlename")) \

.withColumn("lname",col("name.lastname")) \

.drop("name")

df4.printSchema()

## 6. Using col() function – To Dynamically rename all or multiple columns

Another way to change all column names on Dataframe is to use col() function.

IN progress

## 7. Using toDF() – To change all columns in a PySpark DataFrame

When we have data in a flat structure (without nested) , use toDF() with a new schema to change all column names.

newColumns = ["newCol1","newCol2","newCol3","newCol4"]

df.toDF(\*newColumns).printSchema()

## Source code

import pyspark

from pyspark.sql import SparkSession

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

from pyspark.sql.functions import \*

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

dataDF = [(('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)

]

schema = StructType([

StructField('name', StructType([

StructField('firstname', StringType(), True),

StructField('middlename', StringType(), True),

StructField('lastname', StringType(), True)

])),

StructField('dob', StringType(), True),

StructField('gender', StringType(), True),

StructField('salary', IntegerType(), True)

])

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

df.printSchema()

# Example 1

df.withColumnRenamed("dob","DateOfBirth").printSchema()

# Example 2

df2 = df.withColumnRenamed("dob","DateOfBirth") \

.withColumnRenamed("salary","salary\_amount")

df2.printSchema()

# Example 3

schema2 = StructType([

StructField("fname",StringType()),

StructField("middlename",StringType()),

StructField("lname",StringType())])

df.select(col("name").cast(schema2),

col("dob"),

col("gender"),

col("salary")) \

.printSchema()

# Example 4

df.select(col("name.firstname").alias("fname"),

col("name.middlename").alias("mname"),

col("name.lastname").alias("lname"),

col("dob"),col("gender"),col("salary")) \

.printSchema()

# Example 5

df4 = df.withColumn("fname",col("name.firstname")) \

.withColumn("mname",col("name.middlename")) \

.withColumn("lname",col("name.lastname")) \

.drop("name")

df4.printSchema()

#Example 7

newColumns = ["newCol1","newCol2","newCol3","newCol4"]

df.toDF(\*newColumns).printSchema()

# Example 6

'''

not working

old\_columns = Seq("dob","gender","salary","fname","mname","lname")

new\_columns = Seq("DateOfBirth","Sex","salary","firstName","middleName","lastName")

columnsList = old\_columns.zip(new\_columns).map(f=>{col(f.\_1).as(f.\_2)})

df5 = df4.select(columnsList:\_\*)

df5.printSchema()

'''

The complete code can be downloaded from [GitHub](https://github.com/sparkbyexamples/spark-examples/blob/master/spark-sql-examples/src/main/scala/com/sparkbyexamples/spark/dataframe/RenameColDataFrame.scala)

### Conclusion:

This article explains different ways to rename all, a single, multiple, and nested columns on PySpark DataFrame.

# PySpark Where Filter Function | Multiple Conditions

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](https://sparkbyexamples.com/pyspark/pyspark-filter-rows-with-null-values/)

df.filter("state is NULL").show()

df.filter(df.state.isNull()).show()

df.filter(col("state").isNull()).show()

df.filter("state IS NULL AND gender IS NULL").show()

df.filter(df.state.isNull() & df.gender.isNull()).show()

from pyspark.sql.functions import col

df.filter("state IS NOT NULL").show()

df.filter("NOT state IS NULL").show()

df.filter(df.state.isNotNull()).show()

df.filter(col("state").isNotNull()).show()

df.createOrReplaceTempView("DATA")

spark.sql("SELECT \* FROM DATA where STATE IS NULL").show()

spark.sql("SELECT \* FROM DATA where STATE IS NULL AND GENDER IS NULL").show()

spark.sql("SELECT \* FROM DATA where STATE IS NOT NULL").show()

* [Spark Filter – startsWith(), endsWith() Examples](https://sparkbyexamples.com/spark/spark-filter-startswith-endswith-examples/)

import spark.implicits.\_

val data = Seq((1,"James Smith"), (2,"Michael Rose"),

(3,"Robert Williams"), (4,"Rames Rose"),(5,"Rames rose")

)

val df = data.toDF("id","name")

import org.apache.spark.sql.functions.col

df.filter(col("name").startsWith("James")).show()

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

| id| name|

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

| 1|James Smith|

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

df.filter(! col("name").startsWith("James")).show()

df.filter( col("name").startsWith("James") === false).show()

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

| id| name|

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

| 2| Michael Rose|

| 3|Robert Williams|

| 4| Rames Rose|

| 5| Rames rose|

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

df.filter(col("name").endsWith("Rose")).show()

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

| id| name|

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

| 2|Michael Rose|

| 4| Rames Rose|

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

//NOT ends with a string

df.filter(! col("name").endsWith("Rose")).show()

df.filter(col("name").endsWith("Rose")==false).show()

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

| id| name|

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

| 1| James Smith|

| 3|Robert Williams|

| 5| Rames rose|

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

* [Spark Filter – contains(), like(), rlike() Examples](https://sparkbyexamples.com/spark/spark-filter-contains-like-rlike-examples/)

//Filter all rows that contains string 'mes' in a 'name' column

import org.apache.spark.sql.functions.col

df.filter(col("name").contains("mes")).show()

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

| id| name|

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

| 1|James Smith|

| 4| Rames Rose|

| 5| Rames rose|

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

//You can also use with like

df.filter(col("name").like("%mes%")).show()

//Using it on SQL to filter rows

df.createOrReplaceTempView("TAB")

spark.sql("select \* from TAB where name like '%mes%'").show()

#

from pyspark.sql.functions import col

df.filter(col("name").contains("mes")).show()

**Note:** [PySpark Column Functions](https://sparkbyexamples.com/pyspark/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)

Before we start with examples, first let’s [create a DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/). Here, I am using a [DataFrame with StructType](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/) and [ArrayType](https://sparkbyexamples.com/pyspark/pyspark-arraytype-column-with-examples/) 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)

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 |

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

## 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)

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)

## 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()

## 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)

This yields below DataFrame results.

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

|name |languages |state|gender|

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

|[James, , Smith] |[Java, Scala, C++]|OH |M |

|[Mike, Mary, Williams]|[Python, VB] |OH |M |

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

## 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](https://sparkbyexamples.com/pyspark/pyspark-column-functions/) 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()

## 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](https://sparkbyexamples.com/pyspark/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()

## 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|

## 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](https://sparkbyexamples.com/spark/spark-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)

This yields below DataFrame results.

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

|name |languages |state|gender|

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

|[James, , Smith]|[Java, Scala, C++]|OH |M |

|[Anna, Rose, ] |[Spark, Java, C++]|NY |F |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

## 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)

This yields below DataFrame results

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

|name |languages |state|gender|

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

|[Julia, , Williams] |[CSharp, VB]|OH |F |

|[Mike, Mary, Williams]|[Python, VB]|OH |M |

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

## 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)

Examples explained here are also available at [PySpark examples GitHub](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-filter.py) project for reference.

## 11. Conclusion

In this tutorial, I’ve explained how to filter rows from PySpark DataFrame based on single or multiple conditions and SQL expression, also learned filtering rows by providing conditions on the array and struct column with Spark with Python examples.

Alternatively, you can also use where() function to filter the rows on PySpark DataFrame.

# PySpark – Distinct to Drop Duplicate Rows

PySpark distinct() function is used to drop/remove the duplicate rows (all columns) from DataFrame and dropDuplicates() is used to drop rows based on selected (one or multiple) columns. In this article, you will learn how to use distinct() and dropDuplicates() functions with PySpark example.

Before we start, first let’s [create a DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/) with some duplicate rows and values on a few columns. We use this DataFrame to demonstrate how to get distinct multiple columns.

import pyspark

from pyspark.sql import SparkSession

from pyspark.sql.functions import expr

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

data = [("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) \

]

columns= ["employee\_name", "department", "salary"]

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

df.printSchema()

df.show(truncate=False)

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 |

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

On the above table, record with employer name Robert has duplicate rows, As you notice we have 2 rows that have duplicate values on all columns and we have 4 rows that have duplicate values on department and salary columns.

## 1. Get Distinct Rows (By Comparing All Columns)

On the above DataFrame, we have a total of 10 rows with 2 rows having all values duplicated, performing distinct on this DataFrame should get us 9 after removing 1 duplicate row.

distinctDF = df.distinct()

print("Distinct count: "+str(distinctDF.count()))

distinctDF.show(truncate=False)

distinct() function on DataFrame returns a new DataFrame after removing the duplicate records. This example yields the below output.

Distinct count: 9

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

|employee\_name|department|salary|

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

|James |Sales |3000 |

|Michael |Sales |4600 |

|Maria |Finance |3000 |

|Robert |Sales |4100 |

|Saif |Sales |4100 |

|Scott |Finance |3300 |

|Jeff |Marketing |3000 |

|Jen |Finance |3900 |

|Kumar |Marketing |2000 |

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

Alternatively, you can also run dropDuplicates() function which returns a new DataFrame after removing duplicate rows.

df2 = df.dropDuplicates()

print("Distinct count: "+str(df2.count()))

df2.show(truncate=False)

## 2. PySpark Distinct of Selected Multiple Columns

PySpark doesn’t have a distinct method which takes columns that should run distinct on (drop duplicate rows on selected multiple columns) however, it provides another signature of dropDuplicates() function which takes multiple columns to eliminate duplicates.

Note that calling dropDuplicates() on DataFrame returns a new DataFrame with duplicate rows removed.

dropDisDF = df.dropDuplicates(["department","salary"])

print("Distinct count of department & salary : "+str(dropDisDF.count()))

dropDisDF.show(truncate=False)

Yields below output. If you notice the output, It dropped 2 records that are duplicate.

Distinct count of department & salary : 8

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

|employee\_name|department|salary|

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

|Jen |Finance |3900 |

|Maria |Finance |3000 |

|Scott |Finance |3300 |

|Michael |Sales |4600 |

|Kumar |Marketing |2000 |

|Robert |Sales |4100 |

|James |Sales |3000 |

|Jeff |Marketing |3000 |

## 3. Source Code to Get Distinct Rows

import pyspark

from pyspark.sql import SparkSession

from pyspark.sql.functions import expr

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

data = [("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) \

]

columns= ["employee\_name", "department", "salary"]

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

df.printSchema()

df.show(truncate=False)

#Distinct

distinctDF = df.distinct()

print("Distinct count: "+str(distinctDF.count()))

distinctDF.show(truncate=False)

#Drop duplicates

df2 = df.dropDuplicates()

print("Distinct count: "+str(df2.count()))

df2.show(truncate=False)

#Drop duplicates on selected columns

dropDisDF = df.dropDuplicates(["department","salary"])

print("Distinct count of department salary : "+str(dropDisDF.count()))

dropDisDF.show(truncate=False)

}

The complete example is available at [GitHub](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-distinct.py) for reference.

## Conclusion

In this PySpark SQL article, you have learned distinct() method which is used to get the distinct values of rows (all columns) and also learned how to use dropDuplicates() to get the distinct and finally learned using dropDuplicates() function to get distinct of multiple columns.

# PySpark orderBy() and sort() explained

* Post author:[Admin](https://sparkbyexamples.com/author/sparkbyexamples/)
* Post category:[PySpark](https://sparkbyexamples.com/category/pyspark/)

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.

Before we start, first let’s[create a DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/).

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)

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|

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

## 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):

**Example**

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

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

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|

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

## 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)

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)

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|

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

## 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)

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|

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

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)

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)

This complete example is also available at [PySpark sorting GitHub project](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-orderby.py) for reference.

## Conclusion

Here you have learned how to Sort PySpark DataFrame columns using sort(), orderBy() and using SQL sort functions and used this function with PySpark SQL along with Ascending and Descending sorting orders.

# PySpark Groupby Explained with Example

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](https://sparkbyexamples.com/spark/using-groupby-on-dataframe/)

**Syntax:**

groupBy(col1 : scala.Predef.String, cols : scala.Predef.String\*) :

org.apache.spark.sql.RelationalGroupedDataset

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()](https://sparkbyexamples.com/pyspark/pyspark-groupby-explained-with-example/#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](https://sparkbyexamples.com/spark/how-to-pivot-table-and-unpivot-a-spark-dataframe/).

## Preparing Data & creating DataFrame

Before we start, let’s[create the DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/) 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)

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|

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

## 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 |

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

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

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

Calculate the minimum salary of each department using min()

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

Calculate the maximin salary of each department using max()

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

Calculate the average salary of each department using avg()

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

Calculate the mean salary of each department using mean()

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

## 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)

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 |

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

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)

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 |

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

## Using filter on aggregate data

Similar to SQL “HAVING” clause, On PySpark DataFrame we can use either [where()](https://sparkbyexamples.com/pyspark/pyspark-dataframe-filter/) or [filter()](https://sparkbyexamples.com/pyspark/pyspark-dataframe-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)

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 |

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

## 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)

This example is also available at [GitHub PySpark Examples](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-groupby.py) project for reference.

## Conclusion

In this tutorial, you have learned how to use groupBy() and aggregate functions on PySpark DataFrame and also learned how to run these on multiple columns and finally filtering data on the aggregated columns.

Thanks for reading. If you like it, please do share the article by following the below social links and any comments or suggestions are welcome in the comments sections!

# PySpark Join Types | Join Two DataFrames

**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](https://sparkbyexamples.com/spark/spark-shuffle-partitions/).

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.

## 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)

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()](https://sparkbyexamples.com/pyspark/pyspark-dataframe-filter/) and [filter()](https://sparkbyexamples.com/pyspark/pyspark-dataframe-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](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/). 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)

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 |

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

## 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)

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 |

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

## 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)

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 |

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

## 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(false)

empDF.join(deptDF,empDF("emp\_dept\_id") == deptDF("dept\_id"),"leftouter")

.show(false)

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 |

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

## 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)

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 |

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

## 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)

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 |

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

## 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)

Yields below output

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

|emp\_id|name |superior\_emp\_id|year\_joined|emp\_dept\_id|gender|salary|

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

|6 |Brown|2 |2010 |50 | |-1 |

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

## 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)

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 |

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

## 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)

## 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")

## 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)

Examples explained here are available at the [GitHub](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-join.py) project for reference.

### Conclusion

In this PySpark SQL tutorial, you have learned two or more DataFrames can be joined using the join() function of the DataFrame, Join types syntax, usage, and examples with PySpark (Spark with Python), I would also recommend reading through Optimizing SQL Joins to know performance impact on joins.

# PySpark Union and UnionAll Explained

PySpark union() and unionAll() transformations are used to merge two or more DataFrame’s of the same schema or structure. In this PySpark article, I will explain both union transformations with PySpark examples.

**Dataframe union()** – union() method of the DataFrame is used to merge two DataFrame’s of the same structure/schema. If schemas are not the same it returns an error.

**DataFrame unionAll()** – unionAll() is deprecated since Spark “2.0.0” version and replaced with union().

**Note:**In other SQL languages, Union eliminates the duplicates but UnionAll merges two datasets including duplicate records. But, in PySpark both behave the same and recommend using [DataFrame duplicate() function to remove duplicate rows](https://sparkbyexamples.com/spark/spark-remove-duplicate-rows/).

First, let’s create two [DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/) with the same schema.

**First DataFrame**

import pyspark

from pyspark.sql import SparkSession

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) \

]

columns= ["employee\_name","department","state","salary","age","bonus"]

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

df.printSchema()

df.show(truncate=False)

This yields the below schema and DataFrame output.

root

|-- employee\_name: string (nullable = true)

|-- department: string (nullable = true)

|-- state: string (nullable = true)

|-- salary: long (nullable = true)

|-- age: long (nullable = true)

|-- bonus: long (nullable = true)

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

|employee\_name|department|state|salary|age|bonus|

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

|James |Sales |NY |90000 |34 |10000|

|Michael |Sales |NY |86000 |56 |20000|

|Robert |Sales |CA |81000 |30 |23000|

|Maria |Finance |CA |90000 |24 |23000|

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

**Second DataFrame**

Now, let’s create a second Dataframe with the new records and some records from the above Dataframe but with the same schema.

simpleData2 = [("James","Sales","NY",90000,34,10000), \

("Maria","Finance","CA",90000,24,23000), \

("Jen","Finance","NY",79000,53,15000), \

("Jeff","Marketing","CA",80000,25,18000), \

("Kumar","Marketing","NY",91000,50,21000) \

]

columns2= ["employee\_name","department","state","salary","age","bonus"]

df2 = spark.createDataFrame(data = simpleData2, schema = columns2)

df2.printSchema()

df2.show(truncate=False)

This yields below output

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

|employee\_name|department|state|salary|age|bonus|

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

|James |Sales |NY |90000 |34 |10000|

|Maria |Finance |CA |90000 |24 |23000|

|Jen |Finance |NY |79000 |53 |15000|

|Jeff |Marketing |CA |80000 |25 |18000|

|Kumar |Marketing |NY |91000 |50 |21000|

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

## Merge two or more DataFrames using union

DataFrame union() method merges two DataFrames and returns the new DataFrame with all rows from two Dataframes regardless of duplicate data.

unionDF = df.union(df2)

unionDF.show(truncate=False)

As you see below it returns all records.

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

|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|

|James |Sales |NY |90000 |34 |10000|

|Maria |Finance |CA |90000 |24 |23000|

|Jen |Finance |NY |79000 |53 |15000|

|Jeff |Marketing |CA |80000 |25 |18000|

|Kumar |Marketing |NY |91000 |50 |21000|

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

## Merge DataFrames using unionAll

DataFrame unionAll() method is deprecated since PySpark “2.0.0” version and recommends using the union() method.

unionAllDF = df.unionAll(df2)

unionAllDF.show(truncate=False)

Returns the same output as above.

## Merge without Duplicates

Since the union() method returns all rows without distinct records, we will use the distinct() function to return just one record when duplicate exists.

disDF = df.union(df2).distinct()

disDF.show(truncate=False)

Yields below output. As you see, this returns only distinct rows.

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

|employee\_name|department|state|salary|age|bonus|

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

|James |Sales |NY |90000 |34 |10000|

|Maria |Finance |CA |90000 |24 |23000|

|Kumar |Marketing |NY |91000 |50 |21000|

|Michael |Sales |NY |86000 |56 |20000|

|Jen |Finance |NY |79000 |53 |15000|

|Jeff |Marketing |CA |80000 |25 |18000|

|Robert |Sales |CA |81000 |30 |23000|

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

## Complete Example of DataFrame Union

import pyspark

from pyspark.sql import SparkSession

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) \

]

columns= ["employee\_name","department","state","salary","age","bonus"]

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

df.printSchema()

df.show(truncate=False)

simpleData2 = [("James","Sales","NY",90000,34,10000), \

("Maria","Finance","CA",90000,24,23000), \

("Jen","Finance","NY",79000,53,15000), \

("Jeff","Marketing","CA",80000,25,18000), \

("Kumar","Marketing","NY",91000,50,21000) \

]

columns2= ["employee\_name","department","state","salary","age","bonus"]

df2 = spark.createDataFrame(data = simpleData2, schema = columns2)

df2.printSchema()

df2.show(truncate=False)

unionDF = df.union(df2)

unionDF.show(truncate=False)

disDF = df.union(df2).distinct()

disDF.show(truncate=False)

unionAllDF = df.unionAll(df2)

unionAllDF.show(truncate=False)

This complete example is also available at the [GitHub](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-union.py) project.

## Conclusion

In this PySpark article, you have learned how to merge two or more DataFrame’s of the same schema into single DataFrame using Union method and learned the unionAll() is deprecates and use duplicate() to duplicate the same elements.

# Spark Merge Two DataFrames with Different Columns or Schema

In Spark or PySpark let’s see how to merge/union two DataFrames with a different number of columns (different schema). In Spark 3.1, you can easily achieve this using unionByName() transformation by passing allowMissingColumns with the value true. In order version, this property is not available

//Scala

merged\_df = df1.unionByName(df2, true)

#PySpark

merged\_df = df1.unionByName(df2, allowMissingColumns=True)

The difference between unionByName() function and union() is that this function  
resolves columns by name (not by position). In other words, unionByName() is used to merge two DataFrame’s by column names instead of by position.

In case if you are using older than Spark 3.1 version, use below approach to merge DataFrame’s with different column names.

* [Spark Merge DataFrames with Different Columns (Scala Example)](https://sparkbyexamples.com/spark/spark-merge-two-dataframes-with-different-columns/#merge-dataframes-different-columns)
* [PySpark Merge DataFrames with Different Columns (Python Example)](https://sparkbyexamples.com/spark/spark-merge-two-dataframes-with-different-columns/#pyspark-merge-dataframes-different-columns)

## Spark Merge Two DataFrames with Different Columns

In this section I will cover Spark with Scala example of how to merge two different DataFrames, first let’s create DataFrames with different number of columns. DataFrame df1 missing column state and salary and df2 missing column age.

//Create DataFrame df1 with columns name,dept & age

val data = Seq(("James","Sales",34), ("Michael","Sales",56),

("Robert","Sales",30), ("Maria","Finance",24) )

import spark.implicits.\_

val df1 = data.toDF("name","dept","age")

df1.printSchema()

//root

// |-- name: string (nullable = true)

// |-- dept: string (nullable = true)

// |-- age: long (nullable = true)

Second DataFrame

//Create DataFrame df1 with columns name,dep,state & salary

val data2=Seq(("James","Sales","NY",9000),("Maria","Finance","CA",9000),

("Jen","Finance","NY",7900),("Jeff","Marketing","CA",8000))

val df2 = data2.toDF("name","dept","state","salary")

df2.printSchema()

//root

// |-- name: string (nullable = true)

// |-- dept: string (nullable = true)

// |-- state: string (nullable = true)

// |-- salary: long (nullable = true)

Now create a new DataFrames from existing after adding missing columns. newly added columns contains null values and we [add constant column using lit() function](https://sparkbyexamples.com/spark/using-lit-and-typedlit-to-add-a-literal-or-constant-to-spark-dataframe/).

val merged\_cols = df1.columns.toSet ++ df2.columns.toSet

import org.apache.spark.sql.functions.{col,lit}

def getNewColumns(column: Set[String], merged\_cols: Set[String]) = {

merged\_cols.toList.map(x => x match {

case x if column.contains(x) => col(x)

case \_ => lit(null).as(x)

})

}

val new\_df1=df1.select(getNewColumns(df1.columns.toSet, merged\_cols):\_\*)

val new\_df2=df2.select(getNewColumns(df2.columns.toSet, merged\_cols):\_\*)

Finally merge two DataFrame’s by using column names

//Finally join two dataframe's df1 & df2 by name

val merged\_df=new\_df1.unionByName(new\_df2)

merged\_df.show()

//+-------+---------+----+-----+------+

//| name| dept| age|state|salary|

//+-------+---------+----+-----+------+

//| James| Sales| 34| null| null|

//|Michael| Sales| 56| null| null|

//| Robert| Sales| 30| null| null|

//| Maria| Finance| 24| null| null|

//| James| Sales|null| NY| 9000|

//| Maria| Finance|null| CA| 9000|

//| Jen| Finance|null| NY| 7900|

//| Jeff|Marketing|null| CA| 8000|

//+-------+---------+----+-----+------+

## PySpark Merge Two DataFrames with Different Columns

In PySpark to merge two DataFrames with different columns, will use the similar approach explain above and uses unionByName() transformation. First let’s create DataFrame’s with different number of columns.

from pyspark.sql import SparkSession

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

#Create DataFrame df1 with columns name,dept & age

data = [("James","Sales",34), ("Michael","Sales",56), \

("Robert","Sales",30), ("Maria","Finance",24) ]

columns= ["name","dept","age"]

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

df1.printSchema()

#Create DataFrame df1 with columns name,dep,state & salary

data2=[("James","Sales","NY",9000),("Maria","Finance","CA",9000), \

("Jen","Finance","NY",7900),("Jeff","Marketing","CA",8000)]

columns2= ["name","dept","state","salary"]

df2 = spark.createDataFrame(data = data2, schema = columns2)

df2.printSchema()

Now add missing columns ‘state‘ and ‘salary‘ to df1 and ‘age‘ to df2 with null values.

#Add missing columns 'state' & 'salary' to df1

from pyspark.sql.functions import lit

for column in [column for column in df2.columns if column not in df1.columns]:

df1 = df1.withColumn(column, lit(None))

#Add missing column 'age' to df2

for column in [column for column in df1.columns if column not in df2.columns]:

df2 = df2.withColumn(column, lit(None))

Now merge/union the DataFrames using unionByName(). The difference between unionByName() function and union() is that this function  
resolves columns by name (not by position). In other words, unionByName() is used to merge two DataFrame’s by column names instead of by position.

#Finally join two dataframe's df1 & df2 by name

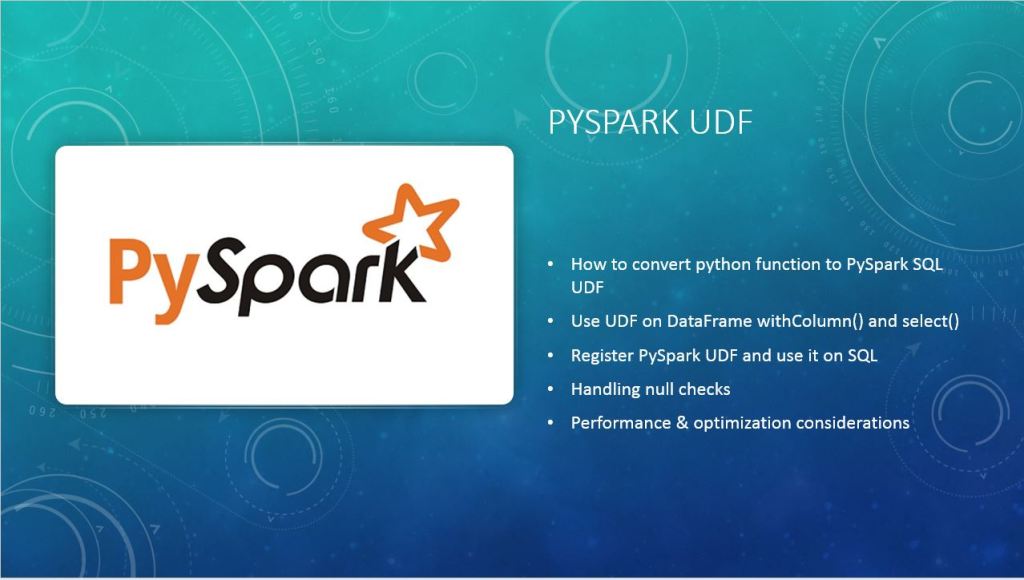
merged\_df=df1.unionByName(df2)

merged\_df.show()

## Conclusion

In this article, you have learned with spark & PySpark examples of how to merge two DataFrames with different columns can be done by adding missing columns to the DataFrame’s and finally union them using unionByName()

# PySpark UDF (User Defined Function)

PySpark UDF Example

PySpark UDF (a.k.a User Defined Function) is the most useful feature of Spark SQL & DataFrame that is used to extend the PySpark build in capabilities. In this article, I will explain what is UDF? why do we need it and how to create and use it on DataFrame select(), [withColumn()](https://sparkbyexamples.com/pyspark/pyspark-dataframe-withcolumn/) and SQL using PySpark (Spark with Python) examples.

**Note:** UDF’s are the most expensive operations hence use them only you have no choice and when essential. In the later section of the article, I will explain why using UDF’s is an expensive operation in detail.

## 1. PySpark UDF Introduction

### 1.1 What is UDF?

UDF’s a.k.a User Defined Functions, If you are coming from SQL background, UDF’s are nothing new to you as most of the traditional RDBMS databases support User Defined Functions, these functions need to register in the database library and use them on SQL as regular functions.

PySpark UDF’s are similar to UDF on traditional databases. In PySpark, you create a function in a Python syntax and wrap it with PySpark SQL udf() or register it as udf and use it on DataFrame and SQL respectively.

### 1.2 Why do we need a UDF?

UDF’s are used to extend the functions of the framework and re-use these functions on multiple DataFrame’s. For example, you wanted to convert every first letter of a word in a name string to a capital case; PySpark build-in features don’t have this function hence you can create it a UDF and reuse this as needed on many Data Frames. UDF’s are once created they can be re-used on several DataFrame’s and SQL expressions.

Before you create any UDF, do your research to check if the similar function you wanted is already available in [Spark SQL Functions](https://sparkbyexamples.com/spark/spark-sql-functions-understanding/). PySpark SQL provides several predefined common functions and many more new functions are added with every release. hence, It is best to check before you reinventing the wheel.

When you creating UDF’s you need to design them very carefully otherwise you will come across optimization & performance issues.

## 2. Create PySpark UDF

### 2.1 Create a DataFrame

Before we jump in creating a UDF, first let’s [create a PySpark DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/).

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

columns = ["Seqno","Name"]

data = [("1", "john jones"),

("2", "tracey smith"),

("3", "amy sanders")]

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

df.show(truncate=False)

Yields below output.

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

|Seqno|Names |

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

|1 |john jones |

|2 |tracey smith|

|3 |amy sanders |

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

### 2.2 Create a Python Function

The first step in creating a UDF is creating a Python function. Below snippet creates a function convertCase() which takes a string parameter and converts the first letter of every word to capital letter. UDF’s take parameters of your choice and returns a value.

def convertCase(str):

resStr=""

arr = str.split(" ")

for x in arr:

resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "

return resStr

### 2.3 Convert a Python function to PySpark UDF

Now convert this function convertCase() to UDF by passing the function to PySpark SQL udf(), this function is available at org.apache.spark.sql.functions.udf package. Make sure you import this package before using it.

PySpark SQL udf() function returns org.apache.spark.sql.expressions.UserDefinedFunction class object.

""" Converting function to UDF """

convertUDF = udf(lambda z: convertCase(z),StringType())

**Note:** The default type of the udf() is StringType hence, you can also write the above statement without return type.

""" Converting function to UDF

StringType() is by default hence not required """

convertUDF = udf(lambda z: convertCase(z))

## 3. Using UDF with DataFrame

### 3.1 Using UDF with PySpark DataFrame select()

Now you can use convertUDF() on a DataFrame column as a regular build-in function.

df.select(col("Seqno"), \

convertUDF(col("Name")).alias("Name") ) \

.show(truncate=False)

This results below output.

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

|Seqno|Name |

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

|1 |John Jones |

|2 |Tracey Smith |

|3 |Amy Sanders |

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

### 3.2 Using UDF with PySpark DataFrame withColumn()

You could also use udf on DataFrame withColumn() function, to explain this I will create another upperCase() function which converts the input string to upper case.

def upperCase(str):

return str.upper()

Let’s convert upperCase() python function to UDF and then use it with DataFrame withColumn(). Below example converts the values of “Name” column to upper case and creates a new column “Curated Name”

upperCaseUDF = udf(lambda z:upperCase(z),StringType())

df.withColumn("Cureated Name", upperCaseUDF(col("Name"))) \

.show(truncate=False)

This yields below output.

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

|Seqno|Name |Cureated Name|

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

|1 |john jones |JOHN JONES |

|2 |tracey smith|TRACEY SMITH |

|3 |amy sanders |AMY SANDERS |

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

### 3.3 Registering PySpark UDF & use it on SQL

In order to use convertCase() function on PySpark SQL, you need to register the function with PySpark by using spark.udf.register().

""" Using UDF on SQL """

spark.udf.register("convertUDF", convertCase,StringType())

df.createOrReplaceTempView("NAME\_TABLE")

spark.sql("select Seqno, convertUDF(Name) as Name from NAME\_TABLE") \

.show(truncate=False)

This yields the same output as 3.1 example.

## 4. Creating UDF using annotation

In the previous sections, you have learned creating a UDF is a 2 step process, first, you need to create a Python function, second convert function to UDF using SQL udf() function, however, you can avoid these two steps and create it with just a single step by using annotations.

@udf(returnType=StringType())

def upperCase(str):

return str.upper()

df.withColumn("Cureated Name", upperCase(col("Name"))) \

.show(truncate=False)

This results same output as section 3.2

## 5. Special Handling

### 5.1 Execution order

One thing to aware is in PySpark/Spark does not guarantee the order of evaluation of subexpressions meaning expressions are not guarantee to evaluated left-to-right or in any other fixed order. PySpark reorders the execution for query optimization and planning hence, AND, OR, WHERE and HAVING expression will have side effects.

So when you are designing and using UDF, you have to be very careful especially with null handling as these results runtime exceptions.

"""

No guarantee Name is not null will execute first

If convertUDF(Name) like '%John%' execute first then

you will get runtime error

"""

spark.sql("select Seqno, convertUDF(Name) as Name from NAME\_TABLE " + \

"where Name is not null and convertUDF(Name) like '%John%'") \

.show(truncate=False)

### 5.2 Handling null check

UDF’s are error-prone when not designed carefully. for example, when you have a column that contains the value null on some records

""" null check """

columns = ["Seqno","Name"]

data = [("1", "john jones"),

("2", "tracey smith"),

("3", "amy sanders"),

('4',None)]

df2 = spark.createDataFrame(data=data,schema=columns)

df2.show(truncate=False)

df2.createOrReplaceTempView("NAME\_TABLE2")

spark.sql("select convertUDF(Name) from NAME\_TABLE2") \

.show(truncate=False)

Note that from the above snippet, record with “Seqno 4” has value “None” for “name” column. Since we are not handling null with UDF function, using this on DataFrame returns below error. Note that in Python None is considered null.

AttributeError: 'NoneType' object has no attribute 'split'

at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.handlePythonException(PythonRunner.scala:456)

at org.apache.spark.sql.execution.python.PythonUDFRunner$$anon$1.read(PythonUDFRunner.scala:81)

at org.apache.spark.sql.execution.python.PythonUDFRunner$$anon$1.read(PythonUDFRunner.scala:64)

at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.hasNext(PythonRunner.scala:410)

at org.apache.spark.InterruptibleIterator.hasNext(InterruptibleIterator.scala:37)

at scala.collection.Iterator$$anon$12.hasNext(Iterator.scala:440)

Below points to remember

* Its always best practice to check for null inside a UDF function rather than checking for null outside.
* In any case, if you can’t do a null check in UDF at lease use IF or CASE WHEN to check for null and call UDF conditionally.

spark.udf.register("\_nullsafeUDF", lambda str: convertCase(str) if not str is None else "" , StringType())

spark.sql("select \_nullsafeUDF(Name) from NAME\_TABLE2") \

.show(truncate=False)

spark.sql("select Seqno, \_nullsafeUDF(Name) as Name from NAME\_TABLE2 " + \

" where Name is not null and \_nullsafeUDF(Name) like '%John%'") \

.show(truncate=False)

This executes successfully without errors as we are checking for null/none while registering UDF.

### 5.3 Performance concern using UDF

UDF’s are a black box to PySpark hence it can’t apply optimization and you will lose all the optimization PySpark does on Dataframe/Dataset. When possible you should use [Spark SQL built-in functions](https://sparkbyexamples.com/spark/spark-sql-functions-understanding/) as these functions provide optimization. Consider creating UDF only when existing built-in SQL function doesn’t have it.

## 6. Complete PySpark UDF Example

Below is complete UDF function example in Scala

import pyspark

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, udf

from pyspark.sql.types import StringType

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

columns = ["Seqno","Name"]

data = [("1", "john jones"),

("2", "tracey smith"),

("3", "amy sanders")]

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

df.show(truncate=False)

def convertCase(str):

resStr=""

arr = str.split(" ")

for x in arr:

resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "

return resStr

""" Converting function to UDF """

convertUDF = udf(lambda z: convertCase(z))

df.select(col("Seqno"), \

convertUDF(col("Name")).alias("Name") ) \

.show(truncate=False)

def upperCase(str):

return str.upper()

upperCaseUDF = udf(lambda z:upperCase(z),StringType())

df.withColumn("Cureated Name", upperCaseUDF(col("Name"))) \

.show(truncate=False)

""" Using UDF on SQL """

spark.udf.register("convertUDF", convertCase,StringType())

df.createOrReplaceTempView("NAME\_TABLE")

spark.sql("select Seqno, convertUDF(Name) as Name from NAME\_TABLE") \

.show(truncate=False)

spark.sql("select Seqno, convertUDF(Name) as Name from NAME\_TABLE " + \

"where Name is not null and convertUDF(Name) like '%John%'") \

.show(truncate=False)

""" null check """

columns = ["Seqno","Name"]

data = [("1", "john jones"),

("2", "tracey smith"),

("3", "amy sanders"),

('4',None)]

df2 = spark.createDataFrame(data=data,schema=columns)

df2.show(truncate=False)

df2.createOrReplaceTempView("NAME\_TABLE2")

spark.udf.register("\_nullsafeUDF", lambda str: convertCase(str) if not str is None else "" , StringType())

spark.sql("select \_nullsafeUDF(Name) from NAME\_TABLE2") \

.show(truncate=False)

spark.sql("select Seqno, \_nullsafeUDF(Name) as Name from NAME\_TABLE2 " + \

" where Name is not null and \_nullsafeUDF(Name) like '%John%'") \

.show(truncate=False)

This example is also available at [Spark GitHub project](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-udf.py) for reference.

## Conclusion

In this article, you have learned the following

* PySpark UDF is a User Defined Function that is used to create a reusable function in Spark.
* Once UDF created, that can be re-used on multiple DataFrames and SQL (after registering).
* The default type of the udf() is StringType.
* You need to handle nulls explicitly otherwise you will see side-effects.

# PySpark map() Transformation

* Post author:[NNK](https://sparkbyexamples.com/author/admin/)
* Post category:[PySpark](https://sparkbyexamples.com/category/pyspark/)

PySpark map (map()) is an RDD transformation that is used to apply the transformation function (lambda) on every element of RDD/DataFrame and returns a new RDD. In this article, you will learn the syntax and usage of the RDD map() transformation with an example and how to use it with DataFrame.

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

* **Note1:** DataFrame doesn’t have map() transformation to use with DataFrame hence you need to DataFrame to RDD first.
* **Note2:** If you have a heavy initialization use PySpark mapPartitions() transformation instead of map(), as with mapPartitions() heavy initialization executes only once for each partition instead of every record.

Related: [Spark map() vs mapPartitions() Explained with Examples](https://sparkbyexamples.com/spark/spark-map-vs-mappartitions-transformation/)

First, let’s create an RDD from the list.

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[1]") \

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

data = ["Project","Gutenberg’s","Alice’s","Adventures",

"in","Wonderland","Project","Gutenberg’s","Adventures",

"in","Wonderland","Project","Gutenberg’s"]

rdd=spark.sparkContext.parallelize(data)

## map() Syntax

map(f, preservesPartitioning=False)

## PySpark map() Example with RDD

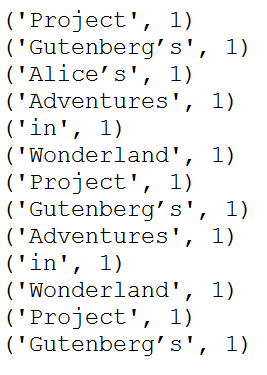
In this PySpark map() example, we are adding a new element with value 1 for each element, the result of the RDD is PairRDDFunctions which contains key-value pairs, word of type String as Key and 1 of type Int as value.

rdd2=rdd.map(lambda x: (x,1))

for element in rdd2.collect():

print(element)

This yields below output.



## PySpark map() Example with DataFrame

PySpark DataFrame doesn’t have map() transformation to apply the lambda function, when you wanted to apply the custom transformation, you need to convert the DataFrame to RDD and apply the map() transformation. Let’s use another dataset to explain this.

data = [('James','Smith','M',30),

('Anna','Rose','F',41),

('Robert','Williams','M',62),

]

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

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

df.show()

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

|firstname|lastname|gender|salary|

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

| James| Smith| M| 30|

| Anna| Rose| F| 41|

| Robert|Williams| M| 62|

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

# Refering columns by index.

rdd2=df.rdd.map(lambda x:

(x[0]+","+x[1],x[2],x[3]\*2)

)

df2=rdd2.toDF(["name","gender","new\_salary"] )

df2.show()

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

| name|gender|new\_salary|

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

| James,Smith| M| 60|

| Anna,Rose| F| 82|

|Robert,Williams| M| 124|

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

Note that aboveI have used index to get the column values, alternatively, you can also refer to the DataFrame column names while iterating.

# Referring Column Names

rdd2=df.rdd.map(lambda x:

(x["firstname"]+","+x["lastname"],x["gender"],x["salary"]\*2)

)

Another alternative

# Referring Column Names

rdd2=df.rdd.map(lambda x:

(x.firstname+","+x.lastname,x.gender,x.salary\*2)

)

You can also create a custom function to perform an operation. Below func1() function executes for every DataFrame row from the lambda function.

# By Calling function

def func1(x):

firstName=x.firstname

lastName=x.lastname

name=firstName+","+lastName

gender=x.gender.lower()

salary=x.salary\*2

return (name,gender,salary)

rdd2=df.rdd.map(lambda x: func1(x))

## Complete PySpark map() example

Below is complete example of PySpark map() transformation.

from pyspark.sql import SparkSession

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

data = ["Project",

"Gutenberg’s",

"Alice’s",

"Adventures",

"in",

"Wonderland",

"Project",

"Gutenberg’s",

"Adventures",

"in",

"Wonderland",

"Project",

"Gutenberg’s"]

rdd=spark.sparkContext.parallelize(data)

rdd2=rdd.map(lambda x: (x,1))

for element in rdd2.collect():

print(element)

data = [('James','Smith','M',30),

('Anna','Rose','F',41),

('Robert','Williams','M',62),

]

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

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

df.show()

rdd2=df.rdd.map(lambda x:

(x[0]+","+x[1],x[2],x[3]\*2)

)

df2=rdd2.toDF(["name","gender","new\_salary"] )

df2.show()

#Referring Column Names

rdd2=df.rdd.map(lambda x:

(x["firstname"]+","+x["lastname"],x["gender"],x["salary"]\*2)

)

#Referring Column Names

rdd2=df.rdd.map(lambda x:

(x.firstname+","+x.lastname,x.gender,x.salary\*2)

)

def func1(x):

firstName=x.firstname

lastName=x.lastname

name=firstName+","+lastName

gender=x.gender.lower()

salary=x.salary\*2

return (name,gender,salary)

rdd2=df.rdd.map(lambda x: func1(x))

In conclusion, you have learned how to apply a map() transformation on every element of PySpark RDD and learned it returns the same number of elements as input RDD. This is one of the differences between map() and <a href="https://sparkbyexamples.com/pyspark/pyspark-rdd-flatmap-transformation/">flatMap()</a> transformations. And you have also learned how to use map() on DataFrame by converting DataFrame to RDD.

Reference: <https://spark.apache.org/docs/latest/api/python/pyspark.sql.html>

# PySpark flatMap() Transformation

PySpark flatMap() is a transformation operation that flattens the RDD/DataFrame (array/map DataFrame columns) after applying the function on every element and returns a new PySpark RDD/DataFrame. In this article, you will learn the syntax and usage of the PySpark flatMap() with an example.

First, let’s create an RDD from the list.

data = ["Project Gutenberg’s",

"Alice’s Adventures in Wonderland",

"Project Gutenberg’s",

"Adventures in Wonderland",

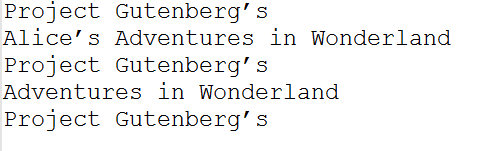
"Project Gutenberg’s"]

rdd=spark.sparkContext.parallelize(data)

for element in rdd.collect():

print(element)

This yields the below output



## flatMap() Syntax

flatMap(f, preservesPartitioning=False)

## flatMap() Example

Now, let’s see with an example of how to apply a flatMap() transformation on 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(" "))

for element in rdd2.collect():

print(element)

This yields below output.

Project

Gutenberg’s

Alice’s

Adventures

in

Wonderland

Project

Gutenberg’s

Adventures

in

Wonderland

Project

Gutenberg’s

## Complete PySpark flatMap() example

Below is the complete example of flatMap() function that works with RDD.

from pyspark.sql import SparkSession

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

data = ["Project Gutenberg’s",

"Alice’s Adventures in Wonderland",

"Project Gutenberg’s",

"Adventures in Wonderland",

"Project Gutenberg’s"]

rdd=spark.sparkContext.parallelize(data)

for element in rdd.collect():

print(element)

#Flatmap

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

for element in rdd2.collect():

print(element)

## Using flatMap() transformation on DataFrame

Unfortunately, PySpark DataFame doesn’t have flatMap() transformation however, DataFrame has [explode() SQL function that is used to flatten the column](https://sparkbyexamples.com/pyspark/pyspark-explode-array-and-map-columns-to-rows/). Below is a complete example.

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'])

from pyspark.sql.functions import explode

df2 = df.select(df.name,explode(df.knownLanguages))

df2.printSchema()

df2.show()

This example flattens the array column “knownLanguages” and yields below output

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|

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

#### Conclusion

In conclusion, you have learned how to apply a PySpark flatMap() transformation to flattens the array or map columns and also learned how to use alternatives for DataFrame.

Reference

* <https://docs.azuredatabricks.net/_static/notebooks/flat-map-transformer-example.html>

# PySpark – Loop/Iterate Through Rows in DataFrame

PySpark provides map(), mapPartitions() to loop/iterate through rows in RDD/DataFrame to perform the complex transformations, and these two returns the same number of records as in the original DataFrame but the number of columns could be different (after add/update).

PySpark also provides foreach() & foreachPartitions() actions to loop/iterate through each Row in a DataFrame but these two returns nothing, In this article, I will explain how to use these methods to get DataFrame column values and process.

## PySpark Loop Through Rows in DataFrame Examples

In order to explain with examples, let’s create a DataFrame

from pyspark.sql import SparkSession

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

data = [('James','Smith','M',30),('Anna','Rose','F',41),

('Robert','Williams','M',62),

]

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

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

df.show()

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

|firstname|lastname|gender|salary|

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

| James| Smith| M| 30|

| Anna| Rose| F| 41|

| Robert|Williams| M| 62|

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

Mostly for simple computations, instead of iterating through using map() and foreach(), you should use either [DataFrame select()](https://sparkbyexamples.com/pyspark/select-columns-from-pyspark-dataframe/) or [DataFrame withColumn()](https://sparkbyexamples.com/pyspark/pyspark-withcolumn/) in conjunction with PySpark SQL functions.

from pyspark.sql.functions import concat\_ws,col,lit

df.select(concat\_ws(",",df.firstname,df.lastname).alias("name"), \

df.gender,lit(df.salary\*2).alias("new\_salary")).show()

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

| name|gender|new\_salary|

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

| James,Smith| M| 60|

| Anna,Rose| F| 82|

|Robert,Williams| M| 124|

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

Below I have map() example to achieve same output as above.

## Using map() to Loop Through Rows in DataFrame

[PySpark map() Transformation](https://sparkbyexamples.com/pyspark/pyspark-map-transformation/) is used to loop/iterate through the PySpark DataFrame/RDD by applying the transformation function (lambda) on every element (Rows and Columns) of RDD/DataFrame. PySpark doesn’t have a map() in DataFrame instead it’s in RDD hence we need to convert DataFrame to RDD first and then use the map(). It returns an RDD and you should [Convert RDD to PySpark DataFrame](https://sparkbyexamples.com/pyspark/convert-pyspark-rdd-to-dataframe/) if needed.

If you have a heavy initialization use PySpark mapPartitions() transformation instead of map(), as with mapPartitions() heavy initialization executes only once for each partition instead of every record.

# Refering columns by index.

rdd=df.rdd.map(lambda x:

(x[0]+","+x[1],x[2],x[3]\*2)

)

df2=rdd.toDF(["name","gender","new\_salary"])

df2.show()

The above example iterates through every row in a DataFrame by applying transformations to the data, since I need a DataFrame back, I have converted the result of RDD to DataFrame with new column names. Note that here I have used index to get the column values, alternatively, you can also refer to the DataFrame column names while iterating.

# Referring Column Names

rdd2=df.rdd.map(lambda x:

(x["firstname"]+","+x["lastname"],x["gender"],x["salary"]\*2)

)

Another alternative

# Referring Column Names

rdd2=df.rdd.map(lambda x:

(x.firstname+","+x.lastname,x.gender,x.salary\*2)

)

You can also create a custom function to perform an operation. Below func1() function executes for every DataFrame row from the lambda function.

# By Calling function

def func1(x):

firstName=x.firstname

lastName=x.lastName

name=firstName+","+lastName

gender=x.gender.lower()

salary=x.salary\*2

return (name,gender,salary)

rdd2=df.rdd.map(lambda x: func1(x))

## Using foreach() to Loop Through Rows in DataFrame

Similar to map(), foreach() also applied to every row of DataFrame, the difference being foreach() is an action and it returns nothing. Below are some examples to iterate through DataFrame using for each.

# Foreach example

def f(x): print(x)

df.foreach(f)

# Another example

df.foreach(lambda x:

print("Data ==>"+x["firstname"]+","+x["lastname"]+","+x["gender"]+","+str(x["salary"]\*2))

)

## Using pandas() to Iterate

If you have a small dataset, you can also [Convert PySpark DataFrame to Pandas](https://sparkbyexamples.com/pyspark/convert-pyspark-dataframe-to-pandas/) and use pandas to iterate through. Use spark.sql.execution.arrow.enabled config to enable Apache Arrow with Spark. Apache Spark uses Apache Arrow which is an in-memory columnar format to transfer the data between Python and JVM.

# Using pandas

import pandas as pd

spark.conf.set("spark.sql.execution.arrow.enabled", "true")

pandasDF = df.toPandas()

for index, row in pandasDF.iterrows():

print(row['firstname'], row['gender'])

## Collect Data As List and Loop Through

You can also [Collect the PySpark DataFrame to Driver](https://sparkbyexamples.com/pyspark/pyspark-collect/) and iterate through Python, you can also use toLocalIterator().

# Collect the data to Python List

dataCollect = df.collect()

for row in dataCollect:

print(row['firstname'] + "," +row['lastname'])

#Using toLocalIterator()

dataCollect=df.rdd.toLocalIterator()

for row in dataCollect:

print(row['firstname'] + "," +row['lastname'])

### Conclusion

In this article, you have learned iterating/loop through Rows of PySpark DataFrame could be done using map(), foreach(), converting to Pandas, and finally converting DataFrame to Python List. If you want to do simile computations, use either select or withColumn().

### References

* <https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.foreach>

# PySpark Random Sample with Example

PySpark provides a pyspark.sql.DataFrame.sample(), pyspark.sql.DataFrame.sampleBy(), RDD.sample(), and RDD.takeSample() methods to get the random sampling subset from the large dataset, In this article I will explain with Python examples .

If you are working as a Data Scientist or Data analyst you often required to analyze a large dataset/file with billions or trillions of records, processing these large datasets takes some time hence during the analysis phase it is recommended to use a random subset sample from the large files.

**Related:** [Spark SQL Sampling with Scala Examples](https://sparkbyexamples.com/spark/spark-sampling-with-examples/)

## 1. PySpark SQL sample() Usage & Examples

PySpark sampling (pyspark.sql.DataFrame.sample()) is a mechanism to get random sample records from the dataset, this is helpful when you have a larger dataset and wanted to analyze/test a subset of the data for example 10% of the original file.

Below is syntax of the sample() function.

sample(withReplacement, fraction, seed=None)

fraction – Fraction of rows to generate, range [0.0, 1.0]. Note that it doesn’t guarantee to provide the exact number of the fraction of records.

seed – Seed for sampling (default a random seed). Used to reproduce the same random sampling.

withReplacement – Sample with replacement or not (default False).

Let’s see some examples.

### 1.1 ****Using****fraction****to get a random sample in PySpark****

By using fraction between 0 to 1, it returns the approximate number of the fraction of the dataset. For example, 0.1 returns 10% of the rows. However, this does not guarantee it returns the exact 10% of the records.

**Note:** If you run these examples on your system, you may see different results.

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.master("local[1]") \

.appName("SparkByExamples.com") \

.getOrCreate()

df=spark.range(100)

print(df.sample(0.06).collect())

//Output: [Row(id=0), Row(id=2), Row(id=17), Row(id=25), Row(id=26), Row(id=44), Row(id=80)]

My DataFrame has 100 records and I wanted to get 6% sample records which are 6 but the sample() function returned 7 records. This proves the sample function doesn’t return the exact fraction specified.

### 1.2 Using seed to reproduce the same Samples in PySpark

Every time you run a sample() function it returns a different set of sampling records, however sometimes during the development and testing phase you may need to regenerate the same sample every time as you need to compare the results from your previous run. To get consistent same random sampling uses the same slice value for every run. Change slice value to get different results.

print(df.sample(0.1,123).collect())

//Output: 36,37,41,43,56,66,69,75,83

print(df.sample(0.1,123).collect())

//Output: 36,37,41,43,56,66,69,75,83

print(df.sample(0.1,456).collect())

//Output: 19,21,42,48,49,50,75,80

On above examples, first 2 I have used slice 123 hence the sampling results are same and for last I have used 456 as slice hence it has returned different sampling records

### 1.3 Sample withReplacement (May contain duplicates)

some times you may need to get a random sample with repeated values. By using the value true, results in repeated values.

print(df.sample(True,0.3,123).collect()) //with Duplicates

//Output: 0,5,9,11,14,14,16,17,21,29,33,41,42,52,52,54,58,65,65,71,76,79,85,96

print(df.sample(0.3,123).collect()) // No duplicates

//Output: 0,4,17,19,24,25,26,36,37,41,43,44,53,56,66,68,69,70,71,75,76,78,83,84,88,94,96,97,98

On first example, values 14, 52 and 65 are repeated values.

### 1.4 Stratified sampling in PySpark

You can get Stratified sampling in PySpark without replacement by using sampleBy() method. It returns a sampling fraction for each stratum. If a stratum is not specified, it takes zero as the default.

**sampleBy() Syntax**

sampleBy(col, fractions, seed=None)

col – column name from DataFrame

fractions – It’s Dictionary type takes key and value.

**sampleBy() Example**

df2=df.select((df.id % 3).alias("key"))

print(df2.sampleBy("key", {0: 0.1, 1: 0.2},0).collect())

//Output: [Row(key=0), Row(key=1), Row(key=1), Row(key=1), Row(key=0), Row(key=1), Row(key=1), Row(key=0), Row(key=1), Row(key=1), Row(key=1)]

## 2. PySpark RDD Sample

PySpark RDD also provides sample() function to get a random sampling, it also has another signature takeSample() that returns an Array[T].

**RDD sample() Syntax & Example**

PySpark RDD sample() function returns the random sampling similar to DataFrame and takes a similar types of parameters but in a different order. Since I’ve already covered the explanation of these parameters on DataFrame, I will not be repeating the explanation on RDD, If not already read I recommend reading the DataFrame section above.

sample() of RDD returns a new RDD by selecting random sampling. Below is a syntax.

sample(self, withReplacement, fraction, seed=None)

Below is an example of RDD sample() function

rdd = spark.sparkContext.range(0,100)

print(rdd.sample(False,0.1,0).collect())

//Output: [24, 29, 41, 64, 86]

print(rdd.sample(True,0.3,123).collect())

//Output: [0, 11, 13, 14, 16, 18, 21, 23, 27, 31, 32, 32, 48, 49, 49, 53, 54, 72, 74, 77, 77, 83, 88, 91, 93, 98, 99]

**RDD takeSample() Syntax & Example**

RDD takeSample() is an action hence you need to careful when you use this function as it returns the selected sample records to driver memory. Returning too much data results in an out-of-memory error similar to [collect()](https://sparkbyexamples.com/spark/spark-dataframe-collect/).

Syntax of RDD takeSample() .

takeSample(self, withReplacement, num, seed=None)

Example of RDD takeSample()

print(rdd.takeSample(False,10,0))

//Output: [58, 1, 96, 74, 29, 24, 32, 37, 94, 91]

print(rdd.takeSample(True,30,123))

//Output: [43, 65, 39, 18, 84, 86, 25, 13, 40, 21, 79, 63, 7, 32, 26, 71, 23, 61, 83, 60, 22, 35, 84, 22, 0, 88, 16, 40, 65, 84]

### Conclusion

In summary, PySpark sampling can be done on RDD and DataFrame. In order to do sampling, you need to know how much data you wanted to retrieve by specifying fractions.

Use seed to regenerate the same sampling multiple times. and

Use withReplacement if you are okay to repeat the random records.

# PySpark fillna() & fill() – Replace NULL/None Values

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

While working on PySpark DataFrame we often need to replace null values since certain operations on null value return error hence, we need to graciously handle nulls as the first step before processing. Also, while writing to a file, it’s always best practice to replace null values, not doing this result nulls on the output file.

As part of the cleanup, sometimes you may need to [Drop Rows with NULL/None Values in PySpark DataFrame](https://sparkbyexamples.com/pyspark/pyspark-drop-rows-with-null-values/) and [Filter Rows by checking IS NULL/NOT NULL](https://sparkbyexamples.com/pyspark/pyspark-filter-rows-with-null-values/) conditions.

In this article, I will use both fill() and fillna() to replace null/none values with an empty string, constant value, and zero(0) on Dataframe columns integer, string with Python examples.

Before we start, Let’s [read a CSV into PySpark DataFrame](https://sparkbyexamples.com/pyspark/pyspark-read-csv-file-into-dataframe/) file, where we have no values on certain rows of String and Integer columns, PySpark assigns null values to these no value columns.

The file we are using here is available at GitHub [small\_zipcode.csv](https://github.com/spark-examples/spark-scala-examples/blob/master/src/main/resources/small_zipcode.csv)

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.master("local[1]") \

.appName("SparkByExamples.com") \

.getOrCreate()

filePath="resources/small\_zipcode.csv"

df = spark.read.options(header='true', inferSchema='true') \

.csv(filePath)

df.printSchema()

df.show(truncate=False)

This yields the below output. As you see columns type, city and population columns have null values.

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

|id |zipcode|type |city |state|population|

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

|1 |704 |STANDARD|null |PR |30100 |

|2 |704 |null |PASEO COSTA DEL SUR|PR |null |

|3 |709 |null |BDA SAN LUIS |PR |3700 |

|4 |76166 |UNIQUE |CINGULAR WIRELESS |TX |84000 |

|5 |76177 |STANDARD|null |TX |null |

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

Now, let’s see how to replace these null values.

## PySpark fillna() & fill() Syntax

PySpark provides [DataFrame.fillna()](https://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame.fillna) and [DataFrameNaFunctions.fill()](https://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.DataFrameNaFunctions.fill) to replace NULL/None values. These two are aliases of each other and returns the same results.

fillna(value, subset=None)

fill(value, subset=None)

* **value** – Value should be the data type of int, long, float, string, or dict. Value specified here will be replaced for NULL/None values.
* **subset** – This is optional, when used it should be the subset of the column names where you wanted to replace NULL/None values.

## PySpark Replace NULL/None Values with Zero (0)

PySpark fill(value:Long) signatures that are available in DataFrameNaFunctions is used to replace NULL/None values with numeric values either zero(0) or any constant value for all integer and long datatype columns of PySpark DataFrame or Dataset.

#Replace 0 for null for all integer columns

df.na.fill(value=0).show()

#Replace Replace 0 for null on only population column

df.na.fill(value=0,subset=["population"]).show()

Above both statements yields the same output, since we have just an integer column population with null values Note that it replaces only Integer columns since our value is 0.

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

|id |zipcode|type |city |state|population|

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

|1 |704 |STANDARD|null |PR |30100 |

|2 |704 |null |PASEO COSTA DEL SUR|PR |0 |

|3 |709 |null |BDA SAN LUIS |PR |3700 |

|4 |76166 |UNIQUE |CINGULAR WIRELESS |TX |84000 |

|5 |76177 |STANDARD|null |TX |0 |

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

## PySpark Replace Null/None Value with Empty String

Now let’s see how to replace NULL/None values with an empty string or any constant values String on all DataFrame String columns.

df.na.fill("").show(false)

Yields below output. This replaces all String type columns with empty/blank string for all NULL values.

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

|id |zipcode|type |city |state|population|

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

|1 |704 |STANDARD| |PR |30100 |

|2 |704 | |PASEO COSTA DEL SUR|PR |null |

|3 |709 | |BDA SAN LUIS |PR |3700 |

|4 |76166 |UNIQUE |CINGULAR WIRELESS |TX |84000 |

|5 |76177 |STANDARD| |TX |null |

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

Now, let’s replace NULL’s on specific columns, below example replace column type with empty string and column city with value “unknown”.

df.na.fill("unknown",["city"]) \

.na.fill("",["type"]).show()

Yields below output. This replaces null values with an empty string for type column and replaces with a constant value “unknown” for city column.

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

|id |zipcode|type |city |state|population|

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

|1 |704 |STANDARD|unknown |PR |30100 |

|2 |704 | |PASEO COSTA DEL SUR|PR |null |

|3 |709 | |BDA SAN LUIS |PR |3700 |

|4 |76166 |UNIQUE |CINGULAR WIRELESS |TX |84000 |

|5 |76177 |STANDARD|unknown |TX |null |

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

Alternatively you can also write the above statement as

df.na.fill({"city": "unknown", "type": ""}) \

.show()

## Complete Code

Below is complete code with Scala example. You can use it by copying it from here or use the GitHub to download the source code.

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.master("local[1]") \

.appName("SparkByExamples.com") \

.getOrCreate()

filePath="resources/small\_zipcode.csv"

df = spark.read.options(header='true', inferSchema='true') \

.csv(filePath)

df.printSchema()

df.show(truncate=False)

df.fillna(value=0).show()

df.fillna(value=0,subset=["population"]).show()

df.na.fill(value=0).show()

df.na.fill(value=0,subset=["population"]).show()

df.fillna(value="").show()

df.na.fill(value="").show()

df.fillna("unknown",["city"]) \

.fillna("",["type"]).show()

df.fillna({"city": "unknown", "type": ""}) \

.show()

df.na.fill("unknown",["city"]) \

.na.fill("",["type"]).show()

df.na.fill({"city": "unknown", "type": ""}) \

.show()

## Conclusion

In this PySpark article, you have learned how to replace null/None values with zero or an empty string on integer and string columns respectively using fill() and fillna() transformation functions.

Thanks for reading. If you recognize my effort or like articles here please do comment or provide any suggestions for improvements in the comments sections!

## Reference:

* <https://spark.apache.org/docs/3.0.0/api/python/pyspark.sql.html>

# PySpark Pivot and Unpivot DataFrame

* Post author:[NNK](https://sparkbyexamples.com/author/admin/)
* Post category:[PySpark](https://sparkbyexamples.com/category/pyspark/)

PySpark pivot() function is used to rotate/transpose the data from one column into multiple Dataframe columns and back using unpivot(). Pivot() It is an aggregation where one of the grouping columns values transposed into individual columns with distinct data.

This tutorial describes and provides a PySpark example on how to create a Pivot table on DataFrame and Unpivot back.

Let’s create a [PySpark DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/) to work with.

data = [("Banana",1000,"USA"), ("Carrots",1500,"USA"), ("Beans",1600,"USA"), \

("Orange",2000,"USA"),("Orange",2000,"USA"),("Banana",400,"China"), \

("Carrots",1200,"China"),("Beans",1500,"China"),("Orange",4000,"China"), \

("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Beans",2000,"Mexico")]

columns= ["Product","Amount","Country"]

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

df.printSchema()

df.show(truncate=False)

DataFrame ‘df’ consists of 3 columns Product, Amount and Country as shown below.

root

|-- Product: string (nullable = true)

|-- Amount: long (nullable = true)

|-- Country: string (nullable = true)

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

|Product|Amount|Country|

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

|Banana |1000 |USA |

|Carrots|1500 |USA |

|Beans |1600 |USA |

|Orange |2000 |USA |

|Orange |2000 |USA |

|Banana |400 |China |

|Carrots|1200 |China |

|Beans |1500 |China |

|Orange |4000 |China |

|Banana |2000 |Canada |

|Carrots|2000 |Canada |

|Beans |2000 |Mexico |

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

## Pivot PySpark DataFrame

PySpark SQL provides pivot() function to rotate the data from one column into multiple columns. It is an aggregation where one of the grouping columns values transposed into individual columns with distinct data. To get the total amount exported to each country of each product, will do group by Product, pivot by Country, and the sum of Amount.

df.groupBy("Product").pivot("Country").sum("Amount")

pivotDF.printSchema()

pivotDF.show(truncate=False)

This will transpose the countries from DataFrame rows into columns and produces below output. where ever data is not present, it represents as null by default.

root

|-- Product: string (nullable = true)

|-- Canada: long (nullable = true)

|-- China: long (nullable = true)

|-- Mexico: long (nullable = true)

|-- USA: long (nullable = true)

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

|Product|Canada|China|Mexico|USA |

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

|Orange |null |4000 |null |4000|

|Beans |null |1500 |2000 |1600|

|Banana |2000 |400 |null |1000|

|Carrots|2000 |1200 |null |1500|

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

## Pivot Performance improvement in PySpark 2.0

version 2.0 on-wards performance has been improved on Pivot, however, if you are using the lower version; note that pivot is a very expensive operation hence, it is recommended to provide column data (if known) as an argument to function as shown below.

countries = ["USA","China","Canada","Mexico"]

pivotDF = df.groupBy("Product").pivot("Country", countries).sum("Amount")

pivotDF.show(truncate=False)

Another approach is to do two-phase aggregation. PySpark 2.0 uses this implementation in order to improve the performance [Spark-13749](https://issues.apache.org/jira/browse/SPARK-13749)

pivotDF = df.groupBy("Product","Country") \

.sum("Amount") \

.groupBy("Product") \

.pivot("Country") \

.sum("sum(Amount)") \

pivotDF.show(truncate=False)

Above two examples returns the same output but with better performance.

## Unpivot PySpark DataFrame

Unpivot is a reverse operation, we can achieve by rotating column values into rows values. PySpark SQL doesn’t have unpivot function hence will use the stack() function. Below code converts column countries to row.

from pyspark.sql.functions import expr

unpivotExpr = "stack(3, 'Canada', Canada, 'China', China, 'Mexico', Mexico) as (Country,Total)"

unPivotDF = pivotDF.select("Product", expr(unpivotExpr)) \

.where("Total is not null")

unPivotDF.show(truncate=False)

unPivotDF.show()

It converts pivoted column “country” to rows.

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

|Product|Country|Total|

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

| Orange| China| 4000|

| Beans| China| 1500|

| Beans| Mexico| 2000|

| Banana| Canada| 2000|

| Banana| China| 400|

|Carrots| Canada| 2000|

|Carrots| China| 1200|

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

## Transpose or Pivot without aggregation

**Can we do PySpark DataFrame transpose or pivot without aggregation?**

off course you can, but unfortunately, you can’t achieve using Pivot function. However, pivoting or transposing DataFrame structure without aggregation from rows to columns and columns to rows can be easily done using PySpark and Scala hack. please refer to [this](https://stackoverflow.com/questions/49392683/transpose-dataframe-without-aggregation-in-spark-with-scala) example.

## Complete Example

The complete code can be downloaded from [GitHub](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-pivot.py)

import pyspark

from pyspark.sql import SparkSession

from pyspark.sql.functions import expr

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

data = [("Banana",1000,"USA"), ("Carrots",1500,"USA"), ("Beans",1600,"USA"), \

("Orange",2000,"USA"),("Orange",2000,"USA"),("Banana",400,"China"), \

("Carrots",1200,"China"),("Beans",1500,"China"),("Orange",4000,"China"), \

("Banana",2000,"Canada"),("Carrots",2000,"Canada"),("Beans",2000,"Mexico")]

columns= ["Product","Amount","Country"]

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

df.printSchema()

df.show(truncate=False)

pivotDF = df.groupBy("Product").pivot("Country").sum("Amount")

pivotDF.printSchema()

pivotDF.show(truncate=False)

pivotDF = df.groupBy("Product","Country") \

.sum("Amount") \

.groupBy("Product") \

.pivot("Country") \

.sum("sum(Amount)")

pivotDF.printSchema()

pivotDF.show(truncate=False)

""" unpivot """

unpivotExpr = "stack(3, 'Canada', Canada, 'China', China, 'Mexico', Mexico) as (Country,Total)"

unPivotDF = pivotDF.select("Product", expr(unpivotExpr)) \

.where("Total is not null")

unPivotDF.show(truncate=False)

#### Conclusion:

We have seen how to Pivot DataFrame with PySpark example and Unpivot it back using SQL functions. And also have seen how PySpark 2.0 changes have improved performance by doing two-phase aggregation.

# PySpark partitionBy() – Write to Disk Example

PySpark partitionBy() is a function of pyspark.sql.DataFrameWriter class which is used to partition the large dataset (DataFrame) into smaller files based on one or multiple columns while writing to disk, let’s see how to use this with Python examples.

Partitioning the data on the file system is a way to improve the performance of the query when dealing with a large dataset in the Data lake. A Data Lake is a centralized repository of structured, semi-structured, unstructured, and binary data that allows you to store a large amount of data as-is in its original raw format.

By following the concepts in this article, it will help you to create an efficient [Data Lake](https://sparkbyexamples.com/data-lake/data-lake-vs-data-warehouse/) for production size data.

## 1. What is PySpark Partition?

PySpark partition is a way to split a large dataset into smaller datasets based on one or more partition keys. When you create a DataFrame from a file/table, based on certain parameters PySpark creates the DataFrame with a certain number of partitions in memory. This is one of the main advantages of PySpark DataFrame over Pandas DataFrame. Transformations on partitioned data run faster as they execute transformations parallelly for each partition.

PySpark supports partition in two ways; partition in memory (DataFrame) and partition on the disk (File system).

**Partition in memory:** You can partition or repartition the DataFrame by calling [repartition() or coalesce()](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/) transformations.

**Partition on disk:** While writing the PySpark DataFrame back to disk, you can choose how to partition the data based on columns using partitionBy() of pyspark.sql.DataFrameWriter. This is similar to [Hives partitions scheme](https://sparkbyexamples.com/apache-hive/hive-partitions-explained-with-examples/).

## 2. Partition Advantages

As you are aware PySpark is designed to process large datasets with 100x faster than the tradition processing, this wouldn’t have been possible with out partition. Below are some of the advantages using PySpark partitions on memory or on disk.

* Fast accessed to the data
* Provides the ability to perform an operation on a smaller dataset

Partition at rest (disk) is a feature of many databases and data processing frameworks and it is key to make jobs work at scale.

## 3. Create DataFrame

Let’s [Create a DataFrame by reading a CSV file](https://sparkbyexamples.com/pyspark/pyspark-read-csv-file-into-dataframe/). You can find the dataset explained in this article at [Github zipcodes.csv file](https://github.com/spark-examples/pyspark-examples/blob/master/resources/simple-zipcodes.csv)

df=spark.read.option("header",True) \

.csv("/tmp/resources/simple-zipcodes.csv")

df.printSchema()

#Display below schema

root

|-- RecordNumber: string (nullable = true)

|-- Country: string (nullable = true)

|-- City: string (nullable = true)

|-- Zipcode: string (nullable = true)

|-- state: string (nullable = true)

From above DataFrame, I will be using state as a partition key for our examples below.

## 4. PySpark partitionBy()

PySpark partitionBy() is a function of pyspark.sql.DataFrameWriter class which is used to partition based on column values while writing DataFrame to Disk/File system.

Syntax: partitionBy(self, \*cols)

When you write PySpark DataFrame to disk by calling partitionBy(), PySpark splits the records based on the partition column and stores each partition data into a sub-directory.

#partitionBy()

df.write.option("header",True) \

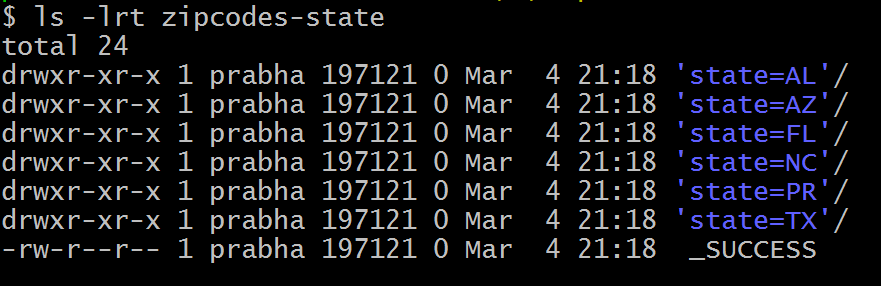
.partitionBy("state") \

.mode("overwrite") \

.csv("/tmp/zipcodes-state")

On our DataFrame, we have a total of 6 different states hence, it creates 6 directories as shown below. The name of the sub-directory would be the partition column and its value (partition column=value).

**Note:** While writing the data as partitions, PySpark eliminates the partition column on the data file and adds partition column & value to the folder name, hence it saves some space on storage.To validate this, open any partition file in a text editor and check.

partitionBy(“state”) example output

On each directory, you may see one or more part files (since our dataset is small, all records for each state are kept in a single part file). You can change this behavior by repartition() the data in memory first. Specify the number of partitions (part files) you would want for each state as an argument to the repartition() method.

## 5. PySpark partitionBy() Multiple Columns

You can also create partitions on multiple columns using PySpark partitionBy(). Just pass columns you want to partition as arguments to this method.

#partitionBy() multiple columns

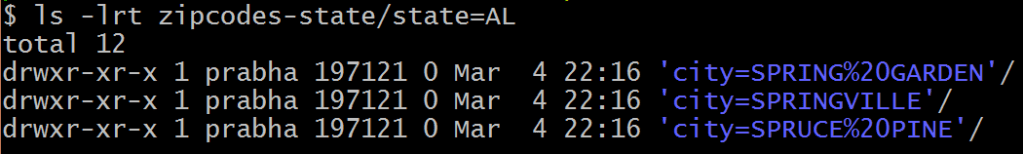
df.write.option("header",True) \

.partitionBy("state","city") \

.mode("overwrite") \

.csv("/tmp/zipcodes-state")

It creates a folder hierarchy for each partition; we have mentioned the first partition as state followed by city hence, it creates a city folder inside the state folder (one folder for each city in a state).

partitonBy(“state”,”city”) multiple columns

## 6. Using repartition() and partitionBy() together

For each partition column, if you wanted to further divide into several partitions, use repartition() and partitionBy() together as explained in the below example.

repartition() creates specified number of partitions in memory. The partitionBy()  will write files to disk for each memory partition and partition column.

#Use repartition() and partitionBy() together

dfRepart.repartition(2)

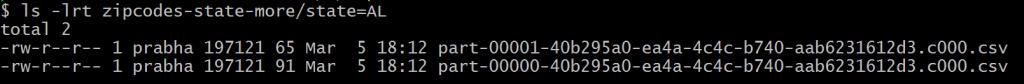
.write.option("header",True) \

.partitionBy("state") \

.mode("overwrite") \

.csv("c:/tmp/zipcodes-state-more")

If you look at the folder, you should see only 2 part files for each state. Dataset has 6 unique states and 2 memory partitions for each state, hence the above code creates a maximum total of 6 x 2 = 12 part files.



**Note:** Since total zipcodes for each US state differ in large, California and Texas have many whereas Delaware has very few, hence it creates a Data Skew (Total rows per each part file differs in large).

## 7. Data Skew – Control Number of Records per Partition File

Use option maxRecordsPerFile if you want to control the number of records for each partition. This is particularly helpful when your data is skewed (Having some partitions with very low records and other partitions with high number of records).

#partitionBy() control number of partitions

df.write.option("header",True) \

.option("maxRecordsPerFile", 2) \

.partitionBy("state") \

.mode("overwrite") \

.csv("/tmp/zipcodes-state")

The above example creates multiple part files for each state and each part file contains just 2 records.

## 8. Read a Specific Partition

Reads are much faster on partitioned data. This code snippet retrieves the data from a specific partition "state=AL and city=SPRINGVILLE". Here, It just reads the data from that specific folder instead of scanning a whole file (when not partitioned).

dfSinglePart=spark.read.option("header",True) \

.csv("c:/tmp/zipcodes-state/state=AL/city=SPRINGVILLE")

dfSinglePart.printSchema()

dfSinglePart.show()

#Displays

root

|-- RecordNumber: string (nullable = true)

|-- Country: string (nullable = true)

|-- Zipcode: string (nullable = true)

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

|RecordNumber|Country|Zipcode|

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

| 54355| US| 35146|

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

While reading specific Partition data into DataFrame, it does not keep the partitions columns on DataFrame hence, you printSchema() and DataFrame is missing state and city columns.

## 9. PySpark SQL – Read Partition Data

This is an example of how to write a Spark DataFrame by preserving the partition columns on DataFrame.

parqDF = spark.read.option("header",True) \

.csv("/tmp/zipcodes-state")

parqDF.createOrReplaceTempView("ZIPCODE")

spark.sql("select \* from ZIPCODE where state='AL' and city = 'SPRINGVILLE'") \

.show()

#Display

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

|RecordNumber|Country|Zipcode|state| city|

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

| 54355| US| 35146| AL|SPRINGVILLE|

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

The execution of this query is also [significantly faster than the query without partition](https://sparkbyexamples.com/spark/spark-performance-tuning/). It filters the data first on state and then applies filters on the city column without scanning the entire dataset.

## 10. How to Choose a Partition Column When Writing to File system?

When creating partitions you have to be very cautious with the number of partitions you would create, as having too many partitions creates too many sub-directories on HDFS which brings unnecessarily and overhead to NameNode (if you are using Hadoop) since it must keep all metadata for the file system in memory.

Let’s assume you have a US census table that contains zip code, city, state, and other columns. Creating a partition on the state, splits the table into around 50 partitions, when searching for a zipcode within a state (state=’CA’ and zipCode =’92704′) results in faster as it needs to scan only in a **state=CA** partition directory.

Partition on zipcode may not be a good option as you might end up with too many partitions.

Another good example of partition is on the Date column. Ideally, you should partition on Year/Month but not on a date.

### Conclusion

While you are create Data Lake out of Azure, HDFS or AWS you need to understand how to partition your data at rest (File system/disk), PySpark partitionBy() and repartition() help you partition the data and eliminating the Data Skew on your large datasets.

Hope this give you better idea on partitions in PySpark.

Happy Learning !!

### References

* <https://spark.apache.org/docs/2.4.0/api/python/pyspark.sql.html?highlight=partition>

# PySpark ArrayType Column With Examples

PySpark pyspark.sql.types.ArrayType (ArrayType extends DataType class) is used to define an array data type column on DataFrame that holds the same type of elements, In this article, I will explain how to create a DataFrame ArrayType column using [org.apache.spark.sql.types.ArrayType](https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/types/ArrayType.scala) class and applying some SQL functions on the array columns with examples.

While working with structured files ([Avro](https://sparkbyexamples.com/spark/read-write-avro-file-spark-dataframe/), [Parquet](https://sparkbyexamples.com/pyspark/pyspark-read-and-write-parquet-file/) e.t.c) or semi-structured ([JSON](https://sparkbyexamples.com/pyspark/pyspark-read-json-file-into-dataframe/)) files, we often get data with complex structures like [MapType](https://sparkbyexamples.com/spark/spark-dataframe-map-maptype-column/), ArrayType, StructType e.t.c. I will try my best to cover some mostly used functions on ArrayType columns.

## What is PySpark ArrayType

PySpark ArrayType is a collection data type that extends the [DataType](https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/types/DataType.scala)class which is a superclass of all types in PySpark. All elements of ArrayType should have the same type of elements.

## Create PySpark ArrayType

You can create an instance of an ArrayType using ArraType() class, This takes arguments valueType and one optional argument valueContainsNull to specify if a value can accept null, by default it takes True. valueType should be a PySpark type that extends DataType class.

from pyspark.sql.types import StringType, ArrayType

arrayCol = ArrayType(StringType(),False)

Above example creates string array and doesn’t not accept null values.

## Create PySpark ArrayType Column Using StructType

Let’s create a DataFrame with few array columns by using [PySpark StructType & StructField classes](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/).

data = [

("James,,Smith",["Java","Scala","C++"],["Spark","Java"],"OH","CA"),

("Michael,Rose,",["Spark","Java","C++"],["Spark","Java"],"NY","NJ"),

("Robert,,Williams",["CSharp","VB"],["Spark","Python"],"UT","NV")

]

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

schema = StructType([

StructField("name",StringType(),True),

StructField("languagesAtSchool",ArrayType(StringType()),True),

StructField("languagesAtWork",ArrayType(StringType()),True),

StructField("currentState", StringType(), True),

StructField("previousState", StringType(), True)

])

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

df.printSchema()

df.show()

This snippet creates two Array columns languagesAtSchool and languagesAtWork which defines languages learned at School and languages using at work. For the rest of the article, I will use these array columns of DataFrame and provide examples of PySpark SQL array functions. printSchema() and show() from above snippet display below output.

root

|-- name: string (nullable = true)

|-- languagesAtSchool: array (nullable = true)

| |-- element: string (containsNull = true)

|-- languagesAtWork: array (nullable = true)

| |-- element: string (containsNull = true)

|-- currentState: string (nullable = true)

|-- previousState: string (nullable = true)

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

| name| languagesAtSchool|languagesAtWork|currentState|previousState|

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

| James,,Smith|[Java, Scala, C++]| [Spark, Java]| OH| CA|

| Michael,Rose,|[Spark, Java, C++]| [Spark, Java]| NY| NJ|

|Robert,,Williams| [CSharp, VB]|[Spark, Python]| UT| NV|

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

## PySpark ArrayType (Array) Functions

[PySpark SQL provides several Array functions](https://sparkbyexamples.com/spark/spark-sql-array-functions/) to work with the ArrayType column, In this section, we will see some of the most commonly used SQL functions.

## explode()

Use explode() function to create a new row for each element in the given array column. There are various [PySpark SQL explode functions](https://sparkbyexamples.com/pyspark/pyspark-explode-array-and-map-columns-to-rows/) available to work with Array columns.

from pyspark.sql.functions import explode

df.select(df.name,explode(df.languagesAtSchool)).show()

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

| name| col|

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

| James,,Smith| Java|

| James,,Smith| Scala|

| James,,Smith| C++|

| Michael,Rose,| Spark|

| Michael,Rose,| Java|

| Michael,Rose,| C++|

|Robert,,Williams|CSharp|

|Robert,,Williams| VB|

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

## Split()

split() sql function returns an array type after splitting the string column by delimiter. Below example split the name column by comma delimiter.

from pyspark.sql.functions import split

df.select(split(df.name,",").alias("nameAsArray")).show()

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

| nameAsArray|

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

| [James, , Smith]|

| [Michael, Rose, ]|

|[Robert, , Williams]|

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

## array()

Use array() function to create a new array column by merging the data from multiple columns. All input columns must have the same data type. The below example combines the data from currentState and previousState and creates a new column states.

from pyspark.sql.functions import array

df.select(df.name,array(df.currentState,df.previousState).alias("States")).show()

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

| name| States|

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

| James,,Smith|[OH, CA]|

| Michael,Rose,|[NY, NJ]|

|Robert,,Williams|[UT, NV]|

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

## array\_contains()

array\_contains() sql function is used to check if array column contains a value. Returns null if the array is null, true if the array contains the value, and false otherwise.

from pyspark.sql.functions import array\_contains

df.select(df.name,array\_contains(df.languagesAtSchool,"Java")

.alias("array\_contains")).show()

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

| name|array\_contains|

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

| James,,Smith| true|

| Michael,Rose,| true|

|Robert,,Williams| false|

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

## Conclusion

You have learned PySpark ArrayType is a collection type similar to an array in other languages that are used to store the same type of elements. ArrayType extends the DataType class (super class of all types) and also learned how to use some commonly used ArrayType functions.

# PySpark MapType (Dict) Usage with Examples

PySpark MapType (also called map type) is a data type to represent Python Dictionary (dict) to store key-value pair, a MapType object comprises three fields, keyType (a DataType), valueType (a DataType) and valueContainsNull (a BooleanType).

**What is PySpark MapType**

PySpark MapType is used to represent map key-value pair similar to python Dictionary (Dict), it extends [DataType](https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/types/DataType.scala) class which is a superclass of all types in PySpark and takes two mandatory arguments keyType and valueType of type DataType and one optional boolean argument valueContainsNull. keyType and valueType can be any type that extends the DataType class. for e.g StringType, IntegerType, ArrayType, MapType, StructType (struct) e.t.c.

## 1. Create PySpark MapType

In order to use MapType data type first, you need to import it from pyspark.sql.types.MapType and use MapType() constructor to create a map object.

from pyspark.sql.types import StringType, MapType

mapCol = MapType(StringType(),StringType(),False)

**MapType Key Points:**

* The First param keyType is used to specify the type of the key in the map.
* The Second param valueType is used to specify the type of the value in the map.
* Third parm valueContainsNull is an optional boolean type that is used to specify if the value of the second param can accept Null/None values.
* The key of the map won’t accept None/Null values.
* PySpark provides several SQL functions to work with MapType.

## 2. Create MapType From StructType

Let’s see how to create a MapType by using [PySpark StructType & StructField](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/), StructType() constructor takes list of StructField, StructField takes a fieldname and type of the value.

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

schema = StructType([

StructField('name', StringType(), True),

StructField('properties', MapType(StringType(),StringType()),True)

])

Now let’s create a DataFrame by using above StructType schema.

from pyspark.sql import SparkSession

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

dataDictionary = [

('James',{'hair':'black','eye':'brown'}),

('Michael',{'hair':'brown','eye':None}),

('Robert',{'hair':'red','eye':'black'}),

('Washington',{'hair':'grey','eye':'grey'}),

('Jefferson',{'hair':'brown','eye':''})

]

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

df.printSchema()

df.show(truncate=False)

Yields below output.

root

|-- Name: string (nullable = true)

|-- properties: map (nullable = true)

| |-- key: string

| |-- value: string (valueContainsNull = true)

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

|Name |properties |

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

|James |[eye -> brown, hair -> black]|

|Michael |[eye ->, hair -> brown] |

|Robert |[eye -> black, hair -> red] |

|Washington|[eye -> grey, hair -> grey] |

|Jefferson |[eye -> , hair -> brown] |

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

## 3. Access PySpark MapType Elements

Let’s see how to extract the key and values from the PySpark DataFrame Dictionary column. Here I have used PySpark map transformation to read the values of properties (MapType column)

df3=df.rdd.map(lambda x: \

(x.name,x.properties["hair"],x.properties["eye"])) \

.toDF(["name","hair","eye"])

df3.printSchema()

df3.show()

root

|-- name: string (nullable = true)

|-- hair: string (nullable = true)

|-- eye: string (nullable = true)

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

| name| hair| eye|

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

| James|black|brown|

| Michael|brown| null|

| Robert| red|black|

|Washington| grey| grey|

| Jefferson|brown| |

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

Let’s use another way to get the value of a key from Map using getItem() of Column type, this method takes a key as an argument and returns a value.

df.withColumn("hair",df.properties.getItem("hair")) \

.withColumn("eye",df.properties.getItem("eye")) \

.drop("properties") \

.show()

df.withColumn("hair",df.properties["hair"]) \

.withColumn("eye",df.properties["eye"]) \

.drop("properties") \

.show()

## 4. Functions

Below are some of the MapType Functions with examples.

### 4.1 – explode

from pyspark.sql.functions import explode

df.select(df.name,explode(df.properties)).show()

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

| name| key|value|

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

| James| eye|brown|

| James|hair|black|

| Michael| eye| null|

| Michael|hair|brown|

| Robert| eye|black|

| Robert|hair| red|

|Washington| eye| grey|

|Washington|hair| grey|

| Jefferson| eye| |

| Jefferson|hair|brown|

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

### 4.2 map\_keys() – Get All Map Keys

from pyspark.sql.functions import map\_keys

df.select(df.name,map\_keys(df.properties)).show()

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

| name|map\_keys(properties)|

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

| James| [eye, hair]|

| Michael| [eye, hair]|

| Robert| [eye, hair]|

|Washington| [eye, hair]|

| Jefferson| [eye, hair]|

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

In case if you wanted to get all map keys as Python List. **WARNING**: **This runs very slow**.

from pyspark.sql.functions import explode,map\_keys

keysDF = df.select(explode(map\_keys(df.properties))).distinct()

keysList = keysDF.rdd.map(lambda x:x[0]).collect()

print(keysList)

#['eye', 'hair']

### 4.3 map\_values() – Get All map Values

from pyspark.sql.functions import map\_values

df.select(df.name,map\_values(df.properties)).show()

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

| name|map\_values(properties)|

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

| James| [brown, black]|

| Michael| [, brown]|

| Robert| [black, red]|

|Washington| [grey, grey]|

| Jefferson| [, brown]|

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

### Conclusion

MapType is a map data structure that is used to store key key-value pairs similar to Python Dictionary (Dic), keys and values type of map should be of a type that extends DataType. Key won’t accept null/None values whereas map of the key can have None/Null value.

# PySpark Aggregate Functions with Examples

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.

## PySpark Aggregate Functions Examples

First, let’s [create a DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/) to work with PySpark aggregate functions. All examples provided here are also available at [PySpark Examples GitHub](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-aggregate.py)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)

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|

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

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

## 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

## 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]|

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

## 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]|

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

## 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]))

## count function

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

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

Prints county: 10

## 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

## 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 |

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

## 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 |

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

## kurtosis function

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

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

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

|kurtosis(salary) |

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

|-0.6467803030303032|

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

## max function

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

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

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

|max(salary)|

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

|4600 |

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

## min function

min() function

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

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

|min(salary)|

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

|2000 |

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

## 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 |

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

## skewness function

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

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

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

|skewness(salary) |

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

|-0.12041791181069571|

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

## 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 |

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

## sum function

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

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

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

|sum(salary)|

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

|34000 |

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

## 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 |

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

## 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 |

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

## 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)

## Conclusion

In this article, I’ve consolidated and listed all PySpark Aggregate functions with scala examples and also learned the benefits of using PySpark SQL functions.

# PySpark Window Functions

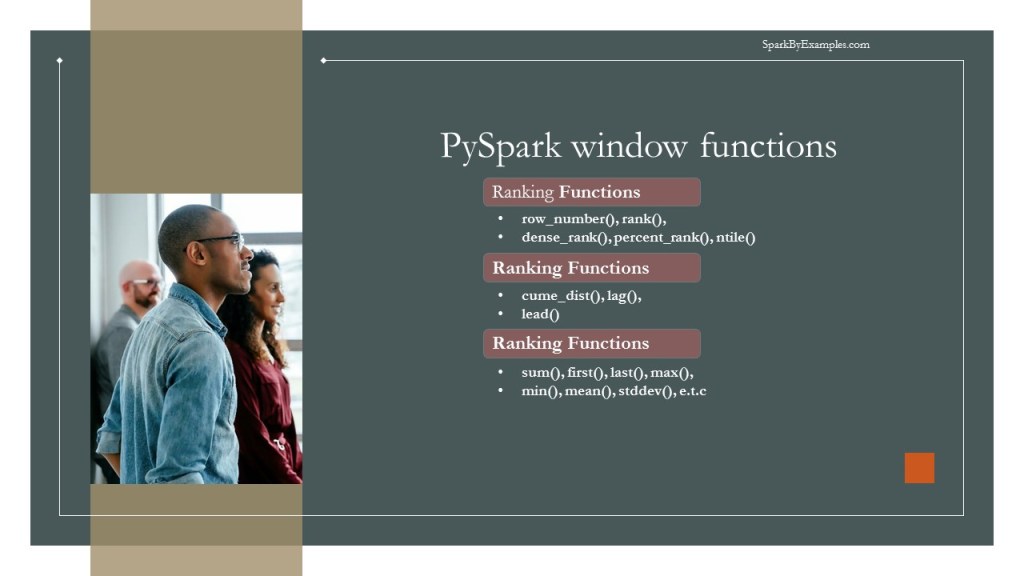
PySpark Window functions are used to calculate results such as the rank, row number e.t.c over a range of input rows. In this article, I’ve explained the concept of window functions, syntax, and finally how to use them with PySpark SQL and PySpark DataFrame API. These come in handy when we need to make aggregate operations in a specific window frame on DataFrame columns.

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.

## 1. Window Functions

PySpark Window functions operate on a group of rows (like frame, partition) and return a single value for every input row. PySpark SQL supports three kinds of window functions:

* [ranking functions](https://sparkbyexamples.com/pyspark/pyspark-window-functions/#ranking-functions)
* [analytic functions](https://sparkbyexamples.com/pyspark/pyspark-window-functions/#analytic-functions)
* [aggregate functions](https://sparkbyexamples.com/pyspark/pyspark-window-functions/#aggregate-functions)

PySpark Window Functions

The below table defines Ranking and Analytic functions and for aggregate functions, we can use any existing [aggregate functions](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/) as a window function.

To perform an operation on a group first, we need to partition the data using Window.partitionBy() , and for row number and rank function we need to additionally order by on partition data using orderBy clause.

Click on each link to know more about these functions along with the Scala examples.

| **WINDOW FUNCTIONS USAGE & SYNTAX** | **PYSPARK WINDOW FUNCTIONS DESCRIPTION** |
| --- | --- |
| row\_number(): Column | Returns a sequential number starting from 1 within a window partition |
| rank(): Column | Returns the rank of rows within a window partition, with gaps. |
| percent\_rank(): Column | Returns the percentile rank of rows within a window partition. |
| dense\_rank(): Column | Returns the rank of rows within a window partition without any gaps. Where as Rank() returns rank with gaps. |
| ntile(n: Int): Column | Returns the ntile id in a window partition |
| cume\_dist(): Column | Returns the cumulative distribution of values within a window partition |
| lag(e: Column, offset: Int): Column lag(columnName: String, offset: Int): Column lag(columnName: String, offset: Int, defaultValue: Any): Column | returns the value that is `offset` rows before the current row, and `null` if there is less than `offset` rows before the current row. |
| lead(columnName: String, offset: Int): Column lead(columnName: String, offset: Int): Column lead(columnName: String, offset: Int, defaultValue: Any): Column | returns the value that is `offset` rows after the current row, and `null` if there is less than `offset` rows after the current row. |

Before we start with an example, first let’s [create a PySpark DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/) to work with.

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) \

)

columns= ["employee\_name", "department", "salary"]

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

df.printSchema()

df.show(truncate=False)

Yields below output

root

|-- employee\_name: string (nullable = true)

|-- department: string (nullable = true)

|-- salary: long (nullable = true)

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

|employee\_name|department|salary|

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

|James |Sales |3000 |

|Michael |Sales |4600 |

|Robert |Sales |4100 |

|Maria |Finance |3000 |

|James |Sales |3000 |

|Scott |Finance |3300 |

|Jen |Finance |3900 |

|Jeff |Marketing |3000 |

|Kumar |Marketing |2000 |

|Saif |Sales |4100 |

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

## 2. PySpark Window ****Ranking functions****

### 2.1 row\_number Window Function

row\_number() window function is used to give the sequential row number starting from 1 to the result of each window partition.

from pyspark.sql.window import Window

from pyspark.sql.functions import row\_number

windowSpec = Window.partitionBy("department").orderBy("salary")

df.withColumn("row\_number",row\_number().over(windowSpec)) \

.show(truncate=False)

Yields below output.

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

|employee\_name|department|salary|row\_number|

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

|James |Sales |3000 |1 |

|James |Sales |3000 |2 |

|Robert |Sales |4100 |3 |

|Saif |Sales |4100 |4 |

|Michael |Sales |4600 |5 |

|Maria |Finance |3000 |1 |

|Scott |Finance |3300 |2 |

|Jen |Finance |3900 |3 |

|Kumar |Marketing |2000 |1 |

|Jeff |Marketing |3000 |2 |

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

### 2.2 rank Window Function

rank() window function is used to provide a rank to the result within a window partition. This function leaves gaps in rank when there are ties.

"""rank"""

from pyspark.sql.functions import rank

df.withColumn("rank",rank().over(windowSpec)) \

.show()

Yields below output.

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

|employee\_name|department|salary|rank|

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

| James| Sales| 3000| 1|

| James| Sales| 3000| 1|

| Robert| Sales| 4100| 3|

| Saif| Sales| 4100| 3|

| Michael| Sales| 4600| 5|

| Maria| Finance| 3000| 1|

| Scott| Finance| 3300| 2|

| Jen| Finance| 3900| 3|

| Kumar| Marketing| 2000| 1|

| Jeff| Marketing| 3000| 2|

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

This is the same as the RANK function in SQL.

### 2.3 dense\_rank Window Function

dense\_rank() window function is used to get the result with rank of rows within a window partition without any gaps. This is similar to rank() function difference being rank function leaves gaps in rank when there are ties.

"""dens\_rank"""

from pyspark.sql.functions import dense\_rank

df.withColumn("dense\_rank",dense\_rank().over(windowSpec)) \

.show()

Yields below output.

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

|employee\_name|department|salary|dense\_rank|

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

| James| Sales| 3000| 1|

| James| Sales| 3000| 1|

| Robert| Sales| 4100| 2|

| Saif| Sales| 4100| 2|

| Michael| Sales| 4600| 3|

| Maria| Finance| 3000| 1|

| Scott| Finance| 3300| 2|

| Jen| Finance| 3900| 3|

| Kumar| Marketing| 2000| 1|

| Jeff| Marketing| 3000| 2|

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

This is the same as the DENSE\_RANK function in SQL.

### 2.4 percent\_rank Window Function

""" percent\_rank """

from pyspark.sql.functions import percent\_rank

df.withColumn("percent\_rank",percent\_rank().over(windowSpec)) \

.show()

Yields below output.

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

|employee\_name|department|salary|percent\_rank|

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

| James| Sales| 3000| 0.0|

| James| Sales| 3000| 0.0|

| Robert| Sales| 4100| 0.5|

| Saif| Sales| 4100| 0.5|

| Michael| Sales| 4600| 1.0|

| Maria| Finance| 3000| 0.0|

| Scott| Finance| 3300| 0.5|

| Jen| Finance| 3900| 1.0|

| Kumar| Marketing| 2000| 0.0|

| Jeff| Marketing| 3000| 1.0|

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

This is the same as the PERCENT\_RANK function in SQL.

### 2.5 ntile Window Function

ntile() window function returns the relative rank of result rows within a window partition. In below example we have used 2 as an argument to ntile hence it returns ranking between 2 values (1 and 2)

"""ntile"""

from pyspark.sql.functions import ntile

df.withColumn("ntile",ntile(2).over(windowSpec)) \

.show()

Yields below output.

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

|employee\_name|department|salary|ntile|

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

| James| Sales| 3000| 1|

| James| Sales| 3000| 1|

| Robert| Sales| 4100| 1|

| Saif| Sales| 4100| 2|

| Michael| Sales| 4600| 2|

| Maria| Finance| 3000| 1|

| Scott| Finance| 3300| 1|

| Jen| Finance| 3900| 2|

| Kumar| Marketing| 2000| 1|

| Jeff| Marketing| 3000| 2|

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

This is the same as the NTILE function in SQL.

## 3. PySpark Window Analytic functions

### 3.1 cume\_dist Window Function

cume\_dist() window function is used to get the cumulative distribution of values within a window partition.

This is the same as the DENSE\_RANK function in SQL.

""" cume\_dist """

from pyspark.sql.functions import cume\_dist

df.withColumn("cume\_dist",cume\_dist().over(windowSpec)) \

.show()

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

|employee\_name|department|salary| cume\_dist|

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

| James| Sales| 3000| 0.4|

| James| Sales| 3000| 0.4|

| Robert| Sales| 4100| 0.8|

| Saif| Sales| 4100| 0.8|

| Michael| Sales| 4600| 1.0|

| Maria| Finance| 3000|0.3333333333333333|

| Scott| Finance| 3300|0.6666666666666666|

| Jen| Finance| 3900| 1.0|

| Kumar| Marketing| 2000| 0.5|

| Jeff| Marketing| 3000| 1.0|

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

### 3.2 lag Window Function

This is the same as the LAG function in SQL.

"""lag"""

from pyspark.sql.functions import lag

df.withColumn("lag",lag("salary",2).over(windowSpec)) \

.show()

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

|employee\_name|department|salary| lag|

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

| James| Sales| 3000|null|

| James| Sales| 3000|null|

| Robert| Sales| 4100|3000|

| Saif| Sales| 4100|3000|

| Michael| Sales| 4600|4100|

| Maria| Finance| 3000|null|

| Scott| Finance| 3300|null|

| Jen| Finance| 3900|3000|

| Kumar| Marketing| 2000|null|

| Jeff| Marketing| 3000|null|

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

### 3.3 lead Window Function

This is the same as the LEAD function in SQL.

"""lead"""

from pyspark.sql.functions import lead

df.withColumn("lead",lead("salary",2).over(windowSpec)) \

.show()

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

|employee\_name|department|salary|lead|

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

| James| Sales| 3000|4100|

| James| Sales| 3000|4100|

| Robert| Sales| 4100|4600|

| Saif| Sales| 4100|null|

| Michael| Sales| 4600|null|

| Maria| Finance| 3000|3900|

| Scott| Finance| 3300|null|

| Jen| Finance| 3900|null|

| Kumar| Marketing| 2000|null|

| Jeff| Marketing| 3000|null|

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

## 4. PySpark Window Aggregate Functions

In this section, I will explain how to calculate sum, min, max for each department using PySpark SQL Aggregate window functions and WindowSpec. When working with Aggregate functions, we don’t need to use order by clause.

windowSpecAgg = Window.partitionBy("department")

from pyspark.sql.functions import col,avg,sum,min,max,row\_number

df.withColumn("row",row\_number().over(windowSpec)) \

.withColumn("avg", avg(col("salary")).over(windowSpecAgg)) \

.withColumn("sum", sum(col("salary")).over(windowSpecAgg)) \

.withColumn("min", min(col("salary")).over(windowSpecAgg)) \

.withColumn("max", max(col("salary")).over(windowSpecAgg)) \

.where(col("row")==1).select("department","avg","sum","min","max") \

.show()

This yields below output

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

|department| avg| sum| min| max|

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

| Sales|3760.0|18800|3000|4600|

| Finance|3400.0|10200|3000|3900|

| Marketing|2500.0| 5000|2000|3000|

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

Please refer for more [Aggregate Functions](https://sparkbyexamples.com/spark/spark-sql-aggregate-functions/)

## Source Code of Window Functions Example

import pyspark

from pyspark.sql import SparkSession

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) \

)

columns= ["employee\_name", "department", "salary"]

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

df.printSchema()

df.show(truncate=False)

from pyspark.sql.window import Window

from pyspark.sql.functions import row\_number

windowSpec = Window.partitionBy("department").orderBy("salary")

df.withColumn("row\_number",row\_number().over(windowSpec)) \

.show(truncate=False)

from pyspark.sql.functions import rank

df.withColumn("rank",rank().over(windowSpec)) \

.show()

from pyspark.sql.functions import dense\_rank

df.withColumn("dense\_rank",dense\_rank().over(windowSpec)) \

.show()

from pyspark.sql.functions import percent\_rank

df.withColumn("percent\_rank",percent\_rank().over(windowSpec)) \

.show()

from pyspark.sql.functions import ntile

df.withColumn("ntile",ntile(2).over(windowSpec)) \

.show()

from pyspark.sql.functions import cume\_dist

df.withColumn("cume\_dist",cume\_dist().over(windowSpec)) \

.show()

from pyspark.sql.functions import lag

df.withColumn("lag",lag("salary",2).over(windowSpec)) \

.show()

from pyspark.sql.functions import lead

df.withColumn("lead",lead("salary",2).over(windowSpec)) \

.show()

windowSpecAgg = Window.partitionBy("department")

from pyspark.sql.functions import col,avg,sum,min,max,row\_number

df.withColumn("row",row\_number().over(windowSpec)) \

.withColumn("avg", avg(col("salary")).over(windowSpecAgg)) \

.withColumn("sum", sum(col("salary")).over(windowSpecAgg)) \

.withColumn("min", min(col("salary")).over(windowSpecAgg)) \

.withColumn("max", max(col("salary")).over(windowSpecAgg)) \

.where(col("row")==1).select("department","avg","sum","min","max") \

.show()

The complete source code is available at [PySpark Examples GitHub](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-window-functions.py) for reference.

## Conclusion

In this tutorial, you have learned what are PySpark SQL Window functions their syntax and how to use them with aggregate function along with several examples in Scala.

References

I would recommend reading [Window Functions Introduction](https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html) and [SQL Window Functions API](https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/functions.scala) blogs for a further understanding of Windows functions. Also, refer to [SQL Window functions](http://www.sqlservertutorial.net/sql-server-window-functions/) to know window functions from native SQL.

# PySpark SQL Date and Timestamp Functions

**PySpark Date and Timestamp Functions** are supported on DataFrame and SQL queries and they work similarly to traditional SQL, Date and Time are very important if you are using PySpark for ETL. Most of all these functions accept input as, Date type, Timestamp type, or String. If a String used, it should be in a default format that can be cast to date.

* DateType default format is yyyy-MM-dd
* TimestampType default format is yyyy-MM-dd HH:mm:ss.SSSS
* Returns null if the input is a string that can not be cast to Date or Timestamp.

PySpark SQL provides several Date & Timestamp functions hence keep an eye on and understand these. Always you should choose these functions instead of writing your own functions (UDF) as these functions are compile-time safe, handles null, and perform better when compared to [PySpark UDF](https://sparkbyexamples.com/pyspark/pyspark-udf-user-defined-function/). If your [PySpark application is critical on performance](https://sparkbyexamples.com/spark/spark-performance-tuning/) try to avoid using custom UDF at all costs as these are not guarantee performance.

For readable purposes, I’ve grouped these functions into the following groups.

* [Date Functions](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#pyspark-sql-date-functions)
* [Timestamp Functions](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#pyspark-sql-timestamp-functions)
* [Date and Timestamp Window Functions](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#sql-date-time)

Before you use any examples below, make sure you [Create PySpark Sparksession](https://sparkbyexamples.com/pyspark/pyspark-what-is-sparksession/) and import SQL functions.

from pyspark.sql.functions import \*

## PySpark SQL Date Functions

Below are some of the PySpark SQL Date functions, these functions operate on the just Date.

The default format of the PySpark Date is yyyy-MM-dd.

| **PYSPARK DATE FUNCTION** | **DATE FUNCTION DESCRIPTION** |
| --- | --- |
| [current\_date()](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#current_date) | Returns the current date as a date column. |
| [date\_format(dateExpr,format)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#date_format) | Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument. |
| [to\_date()](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#to_date) | Converts the column into `DateType` by casting rules to `DateType`. |
| [to\_date(column, fmt)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#to_date) | Converts the column into a `DateType` with a specified format |
| [add\_months(Column, numMonths)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#add_months) | Returns the date that is `numMonths` after `startDate`. |
| [date\_add(column, days)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#add_months) [date\_sub(column, days)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#add_months) | Returns the date that is `days` days after `start` |
| [datediff(end, start)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#datediff) | Returns the number of days from `start` to `end`. |
| [months\_between(end, start)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#months_between) | Returns number of months between dates `start` and `end`. A whole number is returned if both inputs have the same day of month or both are the last day of their respective months. Otherwise, the difference is calculated assuming 31 days per month. |
| [months\_between(end, start, roundOff)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#months_between) | Returns number of months between dates `end` and `start`. If `roundOff` is set to true, the result is rounded off to 8 digits; it is not rounded otherwise. |
| [next\_day(column, dayOfWeek)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#year) | Returns the first date which is later than the value of the `date` column that is on the specified day of the week. For example, `next\_day('2015-07-27', "Sunday")` returns 2015-08-02 because that is the first Sunday after 2015-07-27. |
| [trunc(column, format)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#trunc) | Returns date truncated to the unit specified by the format. For example, `trunc("2018-11-19 12:01:19", "year")` returns 2018-01-01 format: 'year', 'yyyy', 'yy' to truncate by year, 'month', 'mon', 'mm' to truncate by month |
| [date\_trunc(format, timestamp)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#trunc) | Returns timestamp truncated to the unit specified by the format. For example, `date\_trunc("year", "2018-11-19 12:01:19")` returns 2018-01-01 00:00:00 format: 'year', 'yyyy', 'yy' to truncate by year, 'month', 'mon', 'mm' to truncate by month, 'day', 'dd' to truncate by day, Other options are: 'second', 'minute', 'hour', 'week', 'month', 'quarter' |
| [year(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#year) | Extracts the year as an integer from a given date/timestamp/string |
| [quarter(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#year) | Extracts the quarter as an integer from a given date/timestamp/string. |
| [month(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#year) | Extracts the month as an integer from a given date/timestamp/string |
| [dayofweek(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#year) | Extracts the day of the week as an integer from a given date/timestamp/string. Ranges from 1 for a Sunday through to 7 for a Saturday |
| [dayofmonth(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#year) | Extracts the day of the month as an integer from a given date/timestamp/string. |
| [dayofyear(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#year) | Extracts the day of the year as an integer from a given date/timestamp/string. |
| [weekofyear(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#year) | Extracts the week number as an integer from a given date/timestamp/string. A week is considered to start on a Monday and week 1 is the first week with more than 3 days, as defined by ISO 8601 |
| [last\_day(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#year) | Returns the last day of the month which the given date belongs to. For example, input "2015-07-27" returns "2015-07-31" since July 31 is the last day of the month in July 2015. |
| from\_unixtime(column) | Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the yyyy-MM-dd HH:mm:ss format. |
| from\_unixtime(column, f) | Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the given format. |
| unix\_timestamp() | Returns the current Unix timestamp (in seconds) as a long |
| unix\_timestamp(column) | Converts time string in format yyyy-MM-dd HH:mm:ss to Unix timestamp (in seconds), using the default timezone and the default locale. |
| unix\_timestamp(column, p) | Converts time string with given pattern to Unix timestamp (in seconds). |

## PySpark SQL Timestamp Functions

Below are some of the PySpark SQL Timestamp functions, these functions operate on both date and timestamp values.

The default format of the Spark Timestamp is yyyy-MM-dd HH:mm:ss.SSSS

| **PYSPARK TIMESTAMP FUNCTION SIGNATURE** | **TIMESTAMP FUNCTION DESCRIPTION** |
| --- | --- |
| [current\_timestamp ()](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#current_timestamp) | Returns the current timestamp as a timestamp column |
| [hour(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#hour-minute-second) | Extracts the hours as an integer from a given date/timestamp/string. |
| [minute(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#hour-minute-second) | Extracts the minutes as an integer from a given date/timestamp/string. |
| [second(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#hour-minute-second) | Extracts the seconds as an integer from a given date/timestamp/string. |
| [to\_timestamp(column)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#to_timestamp) | Converts to a timestamp by casting rules to `TimestampType`. |
| [to\_timestamp(column, fmt)](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#to_timestamp) | Converts time string with the given pattern to timestamp. |

## Date and Timestamp Window Functions

Below are PySpark Data and Timestamp window functions.

Search:

| **DATE & TIME WINDOW FUNCTION SYNTAX** | **DATE & TIME WINDOW FUNCTION DESCRIPTION** |
| --- | --- |
| window(timeColumn: Column, windowDuration: String, slideDuration: String, startTime: String): Column | Bucketize rows into one or more time windows given a timestamp specifying column. Window starts are inclusive but the window ends are exclusive, e.g. 12:05 will be in the window [12:05,12:10) but not in [12:00,12:05). Windows can support microsecond precision. Windows in the order of months are not supported. |
| window(timeColumn: Column, windowDuration: String, slideDuration: String): Column | Bucketize rows into one or more time windows given a timestamp specifying column. Window starts are inclusive but the window ends are exclusive, e.g. 12:05 will be in the window [12:05,12:10) but not in [12:00,12:05). Windows can support microsecond precision. Windows in the order of months are not supported. The windows start beginning at 1970-01-01 00:00:00 UTC |
| window(timeColumn: Column, windowDuration: String): Column | Generates tumbling time windows given a timestamp specifying column. Window starts are inclusive but the window ends are exclusive, e.g. 12:05 will be in the window [12:05,12:10) but not in [12:00,12:05). Windows can support microsecond precision. Windows in the order of months are not supported. The windows start beginning at 1970-01-01 00:00:00 UTC. |

## PySpark SQL Date and Timestamp Functions Examples

Following are the most used **PySpark S**QL **Date and Timestamp Functions** with examples, you can use these on DataFrame and SQL expressions.

from pyspark.sql import SparkSession

from pyspark.sql.functions import \*

# Create SparkSession

spark = SparkSession.builder \

.appName('SparkByExamples.com') \

.getOrCreate()

data=[["1","2020-02-01"],["2","2019-03-01"],["3","2021-03-01"]]

df=spark.createDataFrame(data,["id","input"])

df.show()

#Result

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

| id|input |

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

| 1|2020-02-01|

| 2|2019-03-01|

| 3|2021-03-01|

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

### current\_date()

Use current\_date() to get the current system date. By default, the data will be returned in yyyy-dd-mm format.

#current\_date()

df.select(current\_date().alias("current\_date")

).show(1)

#Result

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

|current\_date|

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

| 2021-02-22|

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

### date\_format()

The below example uses date\_format() to parses the date and converts from yyyy-dd-mm to MM-dd-yyyy format.

#date\_format()

df.select(col("input"),

date\_format(col("input"), "MM-dd-yyyy").alias("date\_format")

).show()

#Result

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

|input |date\_format|

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

|2020-02-01| 02-01-2020|

|2019-03-01| 03-01-2019|

|2021-03-01| 03-01-2021|

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

### to\_date()

Below example converts string in date format yyyy-MM-dd to a DateType yyyy-MM-dd using to\_date(). You can also use this to convert into any specific format. PySpark supports all patterns supports on Java [DateTimeFormatter](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/time/format/DateTimeFormatter.html).

#to\_date()

df.select(col("input"),

to\_date(col("input"), "yyy-MM-dd").alias("to\_date")

).show()

#Result

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

| input| to\_date|

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

|2020-02-01|2020-02-01|

|2019-03-01|2019-03-01|

|2021-03-01|2021-03-01|

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

### datediff()

The below example returns the difference between two dates using datediff().

#datediff()

df.select(col("input"),

datediff(current\_date(),col("input")).alias("datediff")

).show()

#Result

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

| input|datediff|

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

|2020-02-01| 387|

|2019-03-01| 724|

|2021-03-01| -7|

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

### months\_between()

The below example returns the months between two dates using months\_between().

#months\_between()

df.select(col("input"),

months\_between(current\_date(),col("input")).alias("months\_between")

).show()

#Result

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

| input|months\_between|

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

|2020-02-01| 12.67741935|

|2019-03-01| 23.67741935|

|2021-03-01| -0.32258065|

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

### trunc()

The below example truncates the date at a specified unit using trunc().

#trunc()

df.select(col("input"),

trunc(col("input"),"Month").alias("Month\_Trunc"),

trunc(col("input"),"Year").alias("Month\_Year"),

trunc(col("input"),"Month").alias("Month\_Trunc")

).show()

#Result

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

| input|Month\_Trunc|Month\_Year|Month\_Trunc|

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

|2020-02-01| 2020-02-01|2020-01-01| 2020-02-01|

|2019-03-01| 2019-03-01|2019-01-01| 2019-03-01|

|2021-03-01| 2021-03-01|2021-01-01| 2021-03-01|

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

### add\_months() , date\_add(), date\_sub()

Here we are adding and subtracting date and month from a given input.

#add\_months() , date\_add(), date\_sub()

df.select(col("input"),

add\_months(col("input"),3).alias("add\_months"),

add\_months(col("input"),-3).alias("sub\_months"),

date\_add(col("input"),4).alias("date\_add"),

date\_sub(col("input"),4).alias("date\_sub")

).show()

#Result

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

| input|add\_months|sub\_months| date\_add| date\_sub|

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

|2020-02-01|2020-05-01|2019-11-01|2020-02-05|2020-01-28|

|2019-03-01|2019-06-01|2018-12-01|2019-03-05|2019-02-25|

|2021-03-01|2021-06-01|2020-12-01|2021-03-05|2021-02-25|

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

### year(), month(), month(),next\_day(), weekofyear()

df.select(col("input"),

year(col("input")).alias("year"),

month(col("input")).alias("month"),

next\_day(col("input"),"Sunday").alias("next\_day"),

weekofyear(col("input")).alias("weekofyear")

).show()

#Result

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

| input|year|month| next\_day|weekofyear|

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

|2020-02-01|2020| 2|2020-02-02| 5|

|2019-03-01|2019| 3|2019-03-03| 9|

|2021-03-01|2021| 3|2021-03-07| 9|

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

### dayofweek(), dayofmonth(), dayofyear()

df.select(col("input"),

dayofweek(col("input")).alias("dayofweek"),

dayofmonth(col("input")).alias("dayofmonth"),

dayofyear(col("input")).alias("dayofyear"),

).show()

#Result

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

| input|dayofweek|dayofmonth|dayofyear|

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

|2020-02-01| 7| 1| 32|

|2019-03-01| 6| 1| 60|

|2021-03-01| 2| 1| 60|

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

### current\_timestamp()

Following are the Timestamp Functions that you can use on SQL and on DataFrame. Let’s learn these with examples.

Let’s create a test data.

data=[["1","02-01-2020 11 01 19 06"],["2","03-01-2019 12 01 19 406"],["3","03-01-2021 12 01 19 406"]]

df2=spark.createDataFrame(data,["id","input"])

df2.show(truncate=False)

#Result

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

|id |input |

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

|1 |02-01-2020 11 01 19 06 |

|2 |03-01-2019 12 01 19 406|

|3 |03-01-2021 12 01 19 406|

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

Below example returns the current timestamp in spark default format **yyyy-MM-dd HH:mm:ss**

#current\_timestamp()

df2.select(current\_timestamp().alias("current\_timestamp")

).show(1,truncate=False)

#Result

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

|current\_timestamp |

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

|2021-02-22 20:13:29.673|

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

### to\_timestamp()

Converts string timestamp to Timestamp type format.

#to\_timestamp()

df2.select(col("input"),

to\_timestamp(col("input"), "MM-dd-yyyy HH mm ss SSS").alias("to\_timestamp")

).show(truncate=False)

#Result

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

|input |to\_timestamp |

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

|02-01-2020 11 01 19 06 |2020-02-01 11:01:19.06 |

|03-01-2019 12 01 19 406|2019-03-01 12:01:19.406|

|03-01-2021 12 01 19 406|2021-03-01 12:01:19.406|

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

## hour(), Minute() and second()

#hour, minute,second

data=[["1","2020-02-01 11:01:19.06"],["2","2019-03-01 12:01:19.406"],["3","2021-03-01 12:01:19.406"]]

df3=spark.createDataFrame(data,["id","input"])

df3.select(col("input"),

hour(col("input")).alias("hour"),

minute(col("input")).alias("minute"),

second(col("input")).alias("second")

).show(truncate=False)

#Result

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

|input |hour|minute|second|

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

|2020-02-01 11:01:19.06 |11 |1 |19 |

|2019-03-01 12:01:19.406|12 |1 |19 |

|2021-03-01 12:01:19.406|12 |1 |19 |

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

### Conclusion:

In this post, I’ve consolidated the complete list of Date and Timestamp Functions with a description and example of some commonly used. You can find the complete list on the [following blog](https://databricks.com/blog/2015/09/16/apache-spark-1-5-dataframe-api-highlights.html).

# PySpark JSON Functions with Examples

PySpark JSON functions are used to query or extract the elements from JSON string of DataFrame column by path, convert it to struct, mapt type e.t.c, In this article, I will explain the most used JSON SQL functions with Python examples.

## 1. PySpark JSON Functions

[from\_json()](https://sparkbyexamples.com/pyspark/pyspark-json-functions-with-examples/#from_json) – Converts JSON string into Struct type or Map type.

[to\_json()](https://sparkbyexamples.com/pyspark/pyspark-json-functions-with-examples/#to_json) – Converts MapType or Struct type to JSON string.

[json\_tuple()](https://sparkbyexamples.com/pyspark/pyspark-json-functions-with-examples/#json_tuple) – Extract the Data from JSON and create them as a new columns.

[get\_json\_object()](https://sparkbyexamples.com/pyspark/pyspark-json-functions-with-examples/#get_json_object) – Extracts JSON element from a JSON string based on json path specified.

[schema\_of\_json()](https://sparkbyexamples.com/pyspark/pyspark-json-functions-with-examples/#schema_of_json) – Create schema string from JSON string

### 1.1. Create DataFrame with Column contains JSON String

In order to explain these JSON functions first, let’s create DataFrame with a column contains JSON string.

from pyspark.sql import SparkSession,Row

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

jsonString="""{"Zipcode":704,"ZipCodeType":"STANDARD","City":"PARC PARQUE","State":"PR"}"""

df=spark.createDataFrame([(1, jsonString)],["id","value"])

df.show(truncate=False)

//+---+--------------------------------------------------------------------------+

//|id |value |

//+---+--------------------------------------------------------------------------+

//|1 |{"Zipcode":704,"ZipCodeType":"STANDARD","City":"PARC PARQUE","State":"PR"}|

//+---+--------------------------------------------------------------------------+

## 2. PySpark JSON Functions Examples

## 2.1. from\_json()

PySpark from\_json() function is used to convert JSON string into Struct type or Map type. The below example converts JSON string to Map key-value pair. I will leave it to you to convert to struct type. Refer, [Convert JSON string to Struct type column](https://sparkbyexamples.com/spark/spark-from_json-convert-json-column-to-struct-map-or-multiple-columns/).

#Convert JSON string column to Map type

from pyspark.sql.types import MapType,StringType

from pyspark.sql.functions import from\_json

df2=df.withColumn("value",from\_json(df.value,MapType(StringType(),StringType())))

df2.printSchema()

df2.show(truncate=False)

//root

// |-- id: integer (nullable = false)

// |-- value: map (nullable = true)

// | |-- key: string

// | |-- value: string (valueContainsNull = true)

//+---+---------------------------------------------------------------------------+

//|id |value |

//+---+---------------------------------------------------------------------------+

//|1 |[Zipcode -> 704, ZipCodeType -> STANDARD, City -> PARC PARQUE, State -> PR]|

//+---+---------------------------------------------------------------------------+

## 2.2. to\_json()

to\_json() function is used to convert DataFrame columns MapType or Struct type to JSON string. Here, I am using df2 that created from above from\_json() example.

from pyspark.sql.functions import to\_json,col

df2.withColumn("value",to\_json(col("value"))) \

.show(truncate=False)

//+---+----------------------------------------------------------------------------+

//|id |value |

//+---+----------------------------------------------------------------------------+

//|1 |{"Zipcode":"704","ZipCodeType":"STANDARD","City":"PARC PARQUE","State":"PR"}|

//+---+----------------------------------------------------------------------------+

## 2.3. json\_tuple()

Function json\_tuple() is used the query or extract the elements from JSON column and create the result as a new columns.

from pyspark.sql.functions import json\_tuple

df.select(col("id"),json\_tuple(col("value"),"Zipcode","ZipCodeType","City")) \

.toDF("id","Zipcode","ZipCodeType","City") \

.show(truncate=False)

//+---+-------+-----------+-----------+

//|id |Zipcode|ZipCodeType|City |

//+---+-------+-----------+-----------+

//|1 |704 |STANDARD |PARC PARQUE|

//+---+-------+-----------+-----------+

## 2.4. get\_json\_object()

get\_json\_object() is used to extract the JSON string based on path from the JSON column.

from pyspark.sql.functions import get\_json\_object

df.select(col("id"),get\_json\_object(col("value"),"$.ZipCodeType").alias("ZipCodeType")) \

.show(truncate=False)

//+---+-----------+

//|id |ZipCodeType|

//+---+-----------+

//|1 |STANDARD |

//+---+-----------+

## 2.5. schema\_of\_json()

Use schema\_of\_json() to create schema string from JSON string column.

from pyspark.sql.functions import schema\_of\_json,lit

schemaStr=spark.range(1) \

.select(schema\_of\_json(lit("""{"Zipcode":704,"ZipCodeType":"STANDARD","City":"PARC PARQUE","State":"PR"}"""))) \

.collect()[0][0]

print(schemaStr)

//struct<City:string,State:string,ZipCodeType:string,Zipcode:bigint>

## 3. Complete Example of PySpark JSON Functions

from pyspark.sql import SparkSession,Row

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

jsonString="""{"Zipcode":704,"ZipCodeType":"STANDARD","City":"PARC PARQUE","State":"PR"}"""

df=spark.createDataFrame([(1, jsonString)],["id","value"])

df.show(truncate=False)

#Convert JSON string column to Map type

from pyspark.sql.types import MapType,StringType

from pyspark.sql.functions import from\_json

df2=df.withColumn("value",from\_json(df.value,MapType(StringType(),StringType())))

df2.printSchema()

df2.show(truncate=False)

from pyspark.sql.functions import to\_json,col

df2.withColumn("value",to\_json(col("value"))) \

.show(truncate=False)

from pyspark.sql.functions import json\_tuple

df.select(col("id"),json\_tuple(col("value"),"Zipcode","ZipCodeType","City")) \

.toDF("id","Zipcode","ZipCodeType","City") \

.show(truncate=False)

from pyspark.sql.functions import get\_json\_object

df.select(col("id"),get\_json\_object(col("value"),"$.ZipCodeType").alias("ZipCodeType")) \

.show(truncate=False)

from pyspark.sql.functions import schema\_of\_json,lit

schemaStr=spark.range(1) \

.select(schema\_of\_json(lit("""{"Zipcode":704,"ZipCodeType":"STANDARD","City":"PARC PARQUE","State":"PR"}"""))) \

.collect()[0][0]

print(schemaStr)

### References

* <https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/functions.html>

# PySpark Read CSV file into DataFrame

PySpark provides csv("path") on DataFrameReader to read a CSV file into PySpark DataFrame and dataframeObj.write.csv("path") to save or write to the CSV file. In this tutorial, you will learn how to read a single file, multiple files, all files from a local directory into DataFrame, applying some transformations, and finally writing DataFrame back to CSV file using PySpark example.

PySpark supports reading a CSV file with a pipe, comma, tab, space, or any other delimiter/separator files.

**Note:** PySpark out of the box supports reading files in CSV, JSON, and many more file formats into PySpark DataFrame.

## 1. PySpark Read CSV File into DataFrame

Using csv("path") or format("csv").load("path") of DataFrameReader, you can read a CSV file into a PySpark DataFrame, These methods take a file path to read from as an argument. When you use format("csv") method, you can also specify the Data sources by their fully qualified name, but for built-in sources, you can simply use their short names (csv,json, parquet, jdbc, text e.t.c).

Refer dataset [zipcodes.csv at GitHub](https://github.com/spark-examples/pyspark-examples/blob/master/resources/zipcodes.csv)

val spark = SparkSession.builder().master("local[1]")

.appName("SparkByExamples.com")

.getOrCreate()

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

df.printSchema()

Using fully qualified data source name, you can alternatively do the following.

df = spark.read.format("csv")

.load("/tmp/resources/zipcodes.csv")

// or

df = spark.read.format("org.apache.spark.sql.csv")

.load("/tmp/resources/zipcodes.csv")

df.printSchema()

This example reads the data into DataFrame columns "\_c0" for the first column and "\_c1" for the second and so on. and by default data type for all these columns is treated as String.

root

|-- \_c0: string (nullable = true)

|-- \_c1: string (nullable = true)

|-- \_c2: string (nullable = true)

### 1.1 Using Header Record For Column Names

If you have a header with column names on your input file, you need to explicitly specify True for header option using <a href="#header">option("header",True)</a> not mentioning this, the API treats header as a data record.

df2 = spark.read.option("header",True) \

.csv("/tmp/resources/zipcodes.csv")

As mentioned earlier, PySpark reads all columns as a string (StringType) by default. I will explain in later sections on how to read the schema (inferschema) from the header record and derive the column type based on the data.

### 1.2 Read Multiple CSV Files

Using the read.csv() method you can also read multiple csv files, just pass all file names by separating comma as a path, for example :

df = spark.read.csv("path1,path2,path3")

#### 1.3 Read all CSV Files in a Directory

 We can read all CSV files from a directory into DataFrame just by passing directory as a path to the csv() method.

df = spark.read.csv("Folder path")

## 2. Options While Reading CSV File

PySpark CSV dataset provides multiple options to work with CSV files. Below are some of the most important options explained with examples.

You can either use chaining option(self, key, value) to use multiple options or use alternate options(self, \*\*options) method.

### 2.1 delimiter

delimiter option is used to specify the column delimiter of the CSV file. By default, it is **comma (,)** character, but can be set to any character like **pipe(|)**, **tab (\t)**, **space** using this option.

df3 = spark.read.options(delimiter=',') \

.csv("C:/apps/sparkbyexamples/src/pyspark-examples/resources/zipcodes.csv")

### 2.2 inferSchema

The default value set to this option is False when setting to true it automatically infers column types based on the data. Note that, it requires reading the data one more time to infer the schema.

df4 = spark.read.options(inferSchema='True',delimiter=',') \

.csv("src/main/resources/zipcodes.csv")

Alternatively you can also write this by chaining option() method.

df4 = spark.read.option("inferSchema",True) \

.option("delimiter",",") \

.csv("src/main/resources/zipcodes.csv")

### 2.3 header

This option is used to read the first line of the CSV file as column names. By default the value of this option is False , and all column types are assumed to be a string.

df3 = spark.read.options(header='True', inferSchema='True', delimiter=',') \

.csv("/tmp/resources/zipcodes.csv")

### 2.4 quotes

When you have a column with a delimiter that used to split the columns, use quotes option to specify the quote character, by default it is ” and delimiters inside quotes are ignored. but using this option you can set any character.

### 2.5 nullValues

Using nullValues option you can specify the string in a CSV to consider as null. For example, if you want to consider a date column with a value **"1900-01-01"** set null on DataFrame.

### 2.6 dateFormat

dateFormat option to used to set the format of the input [DateType](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#pyspark-sql-date-functions) and [TimestampType](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/#pyspark-sql-timestamp-functions) columns. Supports all java.text.SimpleDateFormat formats.

**Note:** Besides the above options, PySpark CSV API also supports many other options, [please refer to this article for details](https://docs.databricks.com/data/data-sources/read-csv.html).

## 3. Reading CSV files with a user-specified custom schema

If you know the schema of the file ahead and do not want to use the inferSchema option for column names and types, use [user-defined custom column names and type](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/) using schema option.

schema = StructType() \

.add("RecordNumber",IntegerType(),True) \

.add("Zipcode",IntegerType(),True) \

.add("ZipCodeType",StringType(),True) \

.add("City",StringType(),True) \

.add("State",StringType(),True) \

.add("LocationType",StringType(),True) \

.add("Lat",DoubleType(),True) \

.add("Long",DoubleType(),True) \

.add("Xaxis",IntegerType(),True) \

.add("Yaxis",DoubleType(),True) \

.add("Zaxis",DoubleType(),True) \

.add("WorldRegion",StringType(),True) \

.add("Country",StringType(),True) \

.add("LocationText",StringType(),True) \

.add("Location",StringType(),True) \

.add("Decommisioned",BooleanType(),True) \

.add("TaxReturnsFiled",StringType(),True) \

.add("EstimatedPopulation",IntegerType(),True) \

.add("TotalWages",IntegerType(),True) \

.add("Notes",StringType(),True)

df\_with\_schema = spark.read.format("csv") \

.option("header", True) \

.schema(schema) \

.load("/tmp/resources/zipcodes.csv")

## 4. Applying DataFrame transformations

Once you have created DataFrame from the CSV file, you can apply all transformation and actions DataFrame support. Please refer to the link for more details.

## 5. Write PySpark DataFrame to CSV file

Use the write() method of the PySpark DataFrameWriter object to write PySpark DataFrame to a CSV file.

df.write.option("header",True) \

.csv("/tmp/spark\_output/zipcodes")

### 5.1 Options

While writing a CSV file you can use several options. for example, header to output the DataFrame column names as header record and delimiter to specify the delimiter on the CSV output file.

df2.write.options(header='True', delimiter=',') \

.csv("/tmp/spark\_output/zipcodes")

Other options available quote,escape,nullValue,dateFormat,quoteMode .

### 5.2 Saving modes

PySpark DataFrameWriter also has a method mode() to specify saving mode.

overwrite – mode is used to overwrite the existing file.

append – To add the data to the existing file.

ignore – Ignores write operation when the file already exists.

error – This is a default option when the file already exists, it returns an error.

df2.write.mode('overwrite').csv("/tmp/spark\_output/zipcodes")

//you can also use this

df2.write.format("csv").mode('overwrite').save("/tmp/spark\_output/zipcodes")

## 6. PySpark Read CSV Complete Example

import pyspark

from pyspark.sql import SparkSession

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

from pyspark.sql.types import ArrayType, DoubleType, BooleanType

from pyspark.sql.functions import col,array\_contains

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

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

df.printSchema()

df2 = spark.read.option("header",True) \

.csv("/tmp/resources/zipcodes.csv")

df2.printSchema()

df3 = spark.read.options(header='True', delimiter=',') \

.csv("/tmp/resources/zipcodes.csv")

df3.printSchema()

schema = StructType() \

.add("RecordNumber",IntegerType(),True) \

.add("Zipcode",IntegerType(),True) \

.add("ZipCodeType",StringType(),True) \

.add("City",StringType(),True) \

.add("State",StringType(),True) \

.add("LocationType",StringType(),True) \

.add("Lat",DoubleType(),True) \

.add("Long",DoubleType(),True) \

.add("Xaxis",IntegerType(),True) \

.add("Yaxis",DoubleType(),True) \

.add("Zaxis",DoubleType(),True) \

.add("WorldRegion",StringType(),True) \

.add("Country",StringType(),True) \

.add("LocationText",StringType(),True) \

.add("Location",StringType(),True) \

.add("Decommisioned",BooleanType(),True) \

.add("TaxReturnsFiled",StringType(),True) \

.add("EstimatedPopulation",IntegerType(),True) \

.add("TotalWages",IntegerType(),True) \

.add("Notes",StringType(),True)

df\_with\_schema = spark.read.format("csv") \

.option("header", True) \

.schema(schema) \

.load(/tmp/resources/zipcodes.csv")

df\_with\_schema.printSchema()

df2.write.option("header",True) \

.csv("/tmp/spark\_output/zipcodes123")

## 7. Conclusion:

In this tutorial, you have learned how to read a CSV file, multiple CSV files and all files from a local folder into PySpark DataFrame, using multiple options to change the default behavior and write CSV files back to DataFrame using different save options.

# PySpark Read and Write Parquet File

Pyspark SQL provides methods to read Parquet file into DataFrame and write DataFrame to Parquet files, parquet() function from DataFrameReader and DataFrameWriter are used to read from and write/create a Parquet file respectively. Parquet files maintain the schema along with the data hence it is used to process a structured file.

In this article, I will explain how to read from and write a parquet file and also will explain how to partition the data and retrieve the partitioned data with the help of SQL.

Below are the simple statements on how to write and read parquet files in PySpark which I will explain in detail later sections.

df.write.parquet("/tmp/out/people.parquet")

parDF1=spark.read.parquet("/temp/out/people.parquet")

Before, I explain in detail, first let’s understand What is Parquet file and its advantages over CSV, JSON and other text file formats.

## What is Parquet File?

[Apache Parquet](https://parquet.apache.org/) file is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model, or programming language.

### Advantages:

While querying columnar storage, it skips the nonrelevant data very quickly, making faster query execution. As a result aggregation queries consume less time compared to row-oriented databases.

It is able to support advanced [nested data structures](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/).

Parquet supports efficient compression options and encoding schemes.

Pyspark SQL provides support for both reading and writing Parquet files that automatically capture the schema of the original data, It also reduces data storage by 75% on average. Pyspark by default supports Parquet in its library hence we don’t need to add any dependency libraries.

## Apache Parquet Pyspark Example

Since we don’t have the parquet file, let’s work with writing parquet from a DataFrame. First, create a Pyspark DataFrame from a list of data using spark.createDataFrame() method.

data =[("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)]

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

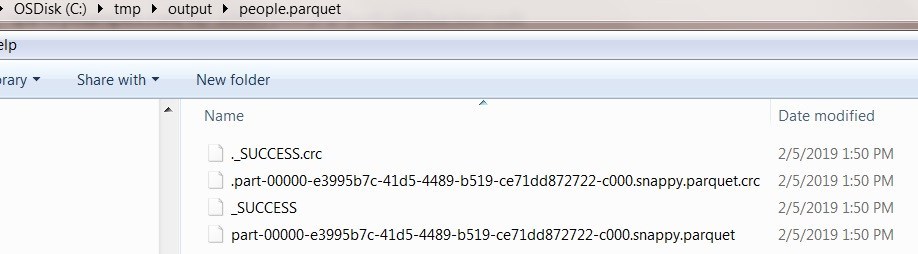
df=spark.createDataFrame(data,columns)

above example, it creates a DataFrame with columns firstname, middlename, lastname, dob, gender, salary.

## Pyspark Write DataFrame to Parquet file format

Now let’s create a parquet file from PySpark DataFrame by calling the parquet() function of DataFrameWriter class. When you write a DataFrame to parquet file, it automatically preserves column names and their data types. Each part file Pyspark creates has the .parquet file extension. Below is the example,

df.write.parquet("/tmp/output/people.parquet")



## Pyspark Read Parquet file into DataFrame

Pyspark provides a parquet() method in DataFrameReader class to read the parquet file into dataframe. Below is an example of a reading parquet file to data frame.

parDF=spark.read.parquet("/tmp/output/people.parquet")

## Append or Overwrite an existing Parquet file

Using append save mode, you can append a dataframe to an existing parquet file. Incase to overwrite use overwrite save mode.

df.write.mode('append').parquet("/tmp/output/people.parquet")

df.write.mode('overwrite').parquet("/tmp/output/people.parquet")

## Executing SQL queries DataFrame

Pyspark Sql provides to create temporary views on parquet files for executing sql queries. These views are available until your program exists.

parqDF.createOrReplaceTempView("ParquetTable")

parkSQL = spark.sql("select \* from ParquetTable where salary >= 4000 ")

## Creating a table on Parquet file

Now let’s walk through executing SQL queries on parquet file. In order to execute sql queries, create a temporary view or table directly on the parquet file instead of creating from DataFrame.

spark.sql("CREATE TEMPORARY VIEW PERSON USING parquet OPTIONS (path \"/tmp/output/people.parquet\")")

spark.sql("SELECT \* FROM PERSON").show()

Here, we created a temporary view PERSON from “people.parquet” file. This gives the following results.

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

|firstname|middlename|lastname| dob|gender|salary|

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

| Robert | |Williams|42114| M| 4000|

| Maria | Anne| Jones|39192| F| 4000|

| Michael | Rose| |40288| M| 4000|

| James | | Smith|36636| M| 3000|

| Jen| Mary| Brown| | F| -1|

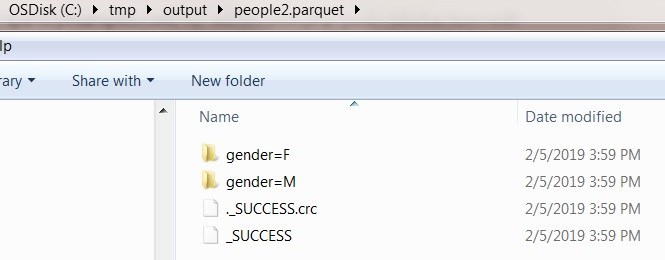
+---------+----------+--------+-----+------+------+

## Create Parquet partition file

When we execute a particular query on the PERSON table, it scan’s through all the rows and returns the results back. This is similar to the traditional database query execution. In PySpark, we can improve query execution in an optimized way by doing partitions on the data using[pyspark partitionBy()](https://sparkbyexamples.com/pyspark/pyspark-partitionby-example/) method. Following is the example of partitionBy().

df.write.partitionBy("gender","salary").mode("overwrite").parquet("/tmp/output/people2.parquet")

When you check the people2.parquet file, it has two partitions “gender” followed by “salary” inside.



## Retrieving from a partitioned Parquet file

The example below explains of reading partitioned parquet file into DataFrame with gender=M.

parDF2=spark.read.parquet("/tmp/output/people2.parquet/gender=M")

parDF2.show(truncate=False)

Output for the above example is shown below.

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

|firstname|middlename|lastname|dob |salary|

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

|Robert | |Williams|42114|4000 |

|Michael |Rose | |40288|4000 |

|James | |Smith |36636|3000 |

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

## Creating a table on Partitioned Parquet file

Here, I am creating a table on partitioned parquet file and executing a query that executes faster than the table without partition, hence improving the performance.

spark.sql("CREATE TEMPORARY VIEW PERSON2 USING parquet OPTIONS (path \"/tmp/output/people2.parquet/gender=F\")")

spark.sql("SELECT \* FROM PERSON2" ).show()

Below is the output .

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

|firstname|middlename|lastname| dob|salary|

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

| Maria | Anne| Jones|39192| 4000|

| Jen| Mary| Brown| | -1|

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

## Complete Example of PySpark read and write Parquet file

import pyspark

from pyspark.sql import SparkSession

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

data =[("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)]

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

df=spark.createDataFrame(data,columns)

df.write.mode("overwrite").parquet("/tmp/output/people.parquet")

parDF1=spark.read.parquet("/tmp/output/people.parquet")

parDF1.createOrReplaceTempView("parquetTable")

parDF1.printSchema()

parDF1.show(truncate=False)

parkSQL = spark.sql("select \* from ParquetTable where salary >= 4000 ")

parkSQL.show(truncate=False)

spark.sql("CREATE TEMPORARY VIEW PERSON USING parquet OPTIONS (path \"/tmp/output/people.parquet\")")

spark.sql("SELECT \* FROM PERSON").show()

df.write.partitionBy("gender","salary").mode("overwrite").parquet("/tmp/output/people2.parquet")

parDF2=spark.read.parquet("/tmp/output/people2.parquet/gender=M")

parDF2.show(truncate=False)

spark.sql("CREATE TEMPORARY VIEW PERSON2 USING parquet OPTIONS (path \"/tmp/output/people2.parquet/gender=F\")")

spark.sql("SELECT \* FROM PERSON2" ).show()

#### Conclusion:

We have learned how to write a Parquet file from a PySpark DataFrame and reading parquet file to DataFrame and created view/tables to execute SQL queries. Also explained how to do partitions on parquet files to [improve performance](https://sparkbyexamples.com/spark/spark-performance-tuning/).

# PySpark Read JSON file into DataFrame

PySpark SQL provides read.json("path") to read a single line or multiline (multiple lines) JSON file into PySpark DataFrame and write.json("path") to save or write to JSON file, In this tutorial, you will learn how to read a single file, multiple files, all files from a directory into DataFrame and writing DataFrame back to JSON file using Python example.

**Related:**

* [PySpark Parse JSON from String Column | TEXT File](https://sparkbyexamples.com/pyspark/pyspark-parse-json-from-string-column-text-file/)
* [Convert JSON Column to Struct, Map or Multiple Columns in PySpark](https://sparkbyexamples.com/spark/spark-from_json-convert-json-column-to-struct-map-or-multiple-columns/)
* [Most used PySpark JSON Functions with Examples](https://sparkbyexamples.com/pyspark/pyspark-json-functions-with-examples/)

**Note:** PySpark API out of the box supports to read JSON files and many more file formats into PySpark DataFrame.

## PySpark Read JSON file into DataFrame

Using read.json("path") or read.format("json").load("path") you can read a JSON file into a PySpark DataFrame, these methods take a file path as an argument.

Unlike [reading a CSV](https://sparkbyexamples.com/pyspark/pyspark-read-csv-file-into-dataframe/), By default JSON data source inferschema from an input file.

[zipcodes.json](https://github.com/spark-examples/pyspark-examples/blob/master/resources/zipcodes.json) file used here can be downloaded from GitHub project.

# Read JSON file into dataframe

df = spark.read.json("resources/zipcodes.json")

df.printSchema()

df.show()

When you use format("json") method, you can also specify the Data sources by their fully qualified name as below.

# Read JSON file into dataframe

df = spark.read.format('org.apache.spark.sql.json') \

.load("resources/zipcodes.json")

## Read JSON file from multiline

PySpark JSON data source provides multiple options to read files in different options, use multiline option to read JSON files scattered across multiple lines. By default multiline option, is set to false.

Below is the input file we going to read, this same file is also available at [Github](https://github.com/spark-examples/pyspark-examples/blob/master/resources/multiline-zipcode.json).

[{

"RecordNumber": 2,

"Zipcode": 704,

"ZipCodeType": "STANDARD",

"City": "PASEO COSTA DEL SUR",

"State": "PR"

},

{

"RecordNumber": 10,

"Zipcode": 709,

"ZipCodeType": "STANDARD",

"City": "BDA SAN LUIS",

"State": "PR"

}]

Using read.option("multiline","true")

# Read multiline json file

multiline\_df = spark.read.option("multiline","true") \

.json("resources/multiline-zipcode.json")

multiline\_df.show()

## Reading multiple files at a time

Using the read.json() method you can also read multiple JSON files from different paths, just pass all file names with fully qualified paths by separating comma, for example

# Read multiple files

df2 = spark.read.json(

['resources/zipcode1.json','resources/zipcode2.json'])

df2.show()

## Reading all files in a directory

We can read all JSON files from a directory into DataFrame just by passing directory as a path to the json() method.

# Read all JSON files from a folder

df3 = spark.read.json("resources/\*.json")

df3.show()

## Reading files with a user-specified custom schema

PySpark Schema defines the structure of the data, in other words, it is the structure of the DataFrame. PySpark SQL provides StructType & StructField classes to programmatically specify the structure to the DataFrame.

If you know the schema of the file ahead and do not want to use the default inferSchema option, use schema option to specify user-defined custom column names and data types.

Use the [PySpark StructType class to create a custom schema](https://sparkbyexamples.com/pyspark/pyspark-structtype-and-structfield/), below we initiate this class and use add a method to add columns to it by providing the column name, data type and nullable option.

# Define custom schema

schema = StructType([

StructField("RecordNumber",IntegerType(),True),

StructField("Zipcode",IntegerType(),True),

StructField("ZipCodeType",StringType(),True),

StructField("City",StringType(),True),

StructField("State",StringType(),True),

StructField("LocationType",StringType(),True),

StructField("Lat",DoubleType(),True),

StructField("Long",DoubleType(),True),

StructField("Xaxis",IntegerType(),True),

StructField("Yaxis",DoubleType(),True),

StructField("Zaxis",DoubleType(),True),

StructField("WorldRegion",StringType(),True),

StructField("Country",StringType(),True),

StructField("LocationText",StringType(),True),

StructField("Location",StringType(),True),

StructField("Decommisioned",BooleanType(),True),

StructField("TaxReturnsFiled",StringType(),True),

StructField("EstimatedPopulation",IntegerType(),True),

StructField("TotalWages",IntegerType(),True),

StructField("Notes",StringType(),True)

])

df\_with\_schema = spark.read.schema(schema) \

.json("resources/zipcodes.json")

df\_with\_schema.printSchema()

df\_with\_schema.show()

## Read JSON file using PySpark SQL

PySpark SQL also provides a way to read a JSON file by creating a temporary view directly from the reading file using spark.sqlContext.sql(“load JSON to temporary view”)

spark.sql("CREATE OR REPLACE TEMPORARY VIEW zipcode USING json OPTIONS" +

" (path 'resources/zipcodes.json')")

spark.sql("select \* from zipcode").show()

## Options while reading JSON file

### nullValues

Using nullValues option you can specify the string in a JSON to consider as null. For example, if you want to consider a date column with a value “1900-01-01” set null on DataFrame.

### dateFormat

dateFormat option to used to set the format of the input DateType and TimestampType columns. Supports all [java.text.SimpleDateFormat](https://docs.oracle.com/javase/10/docs/api/java/time/format/DateTimeFormatter.html) formats.

**Note:** Besides the above options, PySpark JSON dataset also supports many other options.

## Applying DataFrame transformations

Once you have [create PySpark DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/) from the JSON file, you can apply all transformation and actions DataFrame support. Please refer to the link for more details.

## Write PySpark DataFrame to JSON file

Use the PySpark DataFrameWriter object “write” method on DataFrame to write a JSON file.

df2.write.json("/tmp/spark\_output/zipcodes.json")

### PySpark Options while writing JSON files

While writing a JSON file you can use several options.

Other options available nullValue,dateFormat

### PySpark Saving modes

PySpark DataFrameWriter also has a method mode() to specify SaveMode; the argument to this method either takes overwrite, append, ignore, errorifexists.

overwrite – mode is used to overwrite the existing file

append – To add the data to the existing file

ignore – Ignores write operation when the file already exists

errorifexists or error – This is a default option when the file already exists, it returns an error

df2.write.mode('Overwrite').json("/tmp/spark\_output/zipcodes.json")

## Source code for reference

This example is also available at [GitHub PySpark Example Project](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-read-json.py) for reference.

from pyspark.sql import SparkSession

from pyspark.sql.types import StructType,StructField, StringType, IntegerType,BooleanType,DoubleType

spark = SparkSession.builder \

.master("local[1]") \

.appName("SparkByExamples.com") \

.getOrCreate()

# Read JSON file into dataframe

df = spark.read.json("resources/zipcodes.json")

df.printSchema()

df.show()

# Read multiline json file

multiline\_df = spark.read.option("multiline","true") \

.json("resources/multiline-zipcode.json")

multiline\_df.show()

#Read multiple files

df2 = spark.read.json(

['resources/zipcode2.json','resources/zipcode1.json'])

df2.show()

#Read All JSON files from a directory

df3 = spark.read.json("resources/\*.json")

df3.show()

# Define custom schema

schema = StructType([

StructField("RecordNumber",IntegerType(),True),

StructField("Zipcode",IntegerType(),True),

StructField("ZipCodeType",StringType(),True),

StructField("City",StringType(),True),

StructField("State",StringType(),True),

StructField("LocationType",StringType(),True),

StructField("Lat",DoubleType(),True),

StructField("Long",DoubleType(),True),

StructField("Xaxis",IntegerType(),True),

StructField("Yaxis",DoubleType(),True),

StructField("Zaxis",DoubleType(),True),

StructField("WorldRegion",StringType(),True),

StructField("Country",StringType(),True),

StructField("LocationText",StringType(),True),

StructField("Location",StringType(),True),

StructField("Decommisioned",BooleanType(),True),

StructField("TaxReturnsFiled",StringType(),True),

StructField("EstimatedPopulation",IntegerType(),True),

StructField("TotalWages",IntegerType(),True),

StructField("Notes",StringType(),True)

])

df\_with\_schema = spark.read.schema(schema) \

.json("resources/zipcodes.json")

df\_with\_schema.printSchema()

df\_with\_schema.show()

# Create a table from Parquet File

spark.sql("CREATE OR REPLACE TEMPORARY VIEW zipcode3 USING json OPTIONS" +

" (path 'resources/zipcodes.json')")

spark.sql("select \* from zipcode3").show()

# PySpark write Parquet File

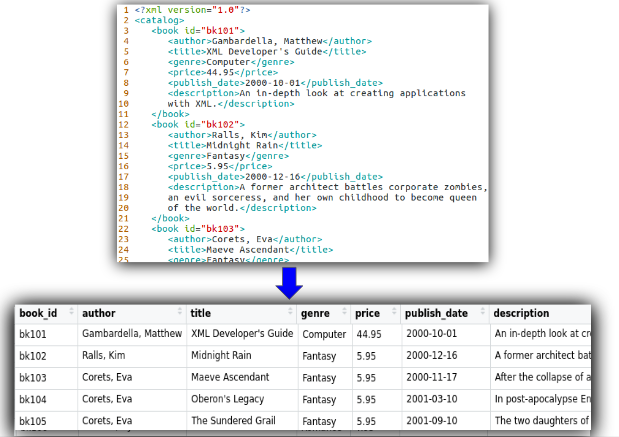
df2.write.mode('Overwrite').json("/tmp/spark\_output/zipcodes.json")

## Conclusion:

In this tutorial, you have learned how to read a JSON file with single line record and multiline record into PySpark DataFrame, and also learned reading single and multiple files at a time and writing JSON file back to DataFrame using different save options.

# Spark Read XML file

**Introduction**

In this post, we are going to use PySpark to process xml files to extract the required records, transform them into DataFrame, then write as csv files (or any other format) to the destination. The input and the output of this task looks like below.

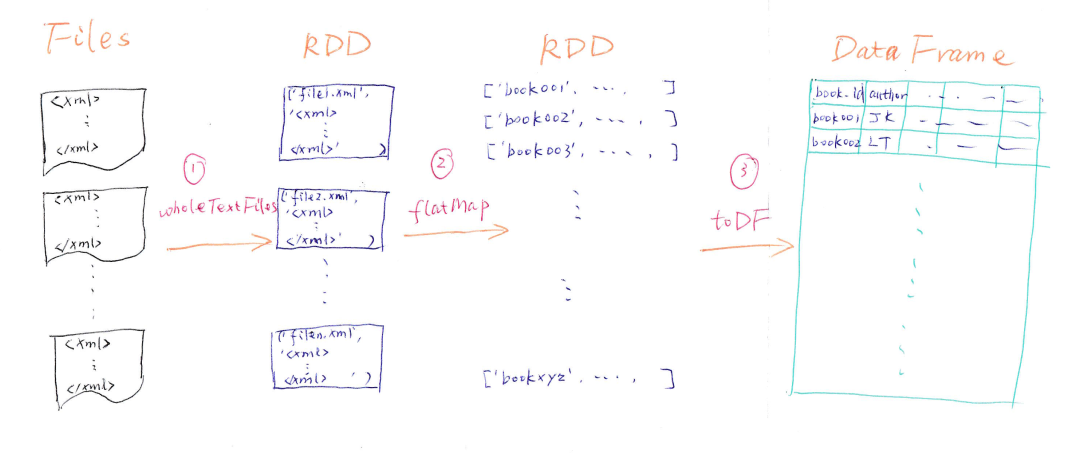
**XML files**

[XML](https://www.w3schools.com/xml/xml_whatis.asp) is designed to store and transport data. XML is self-descriptive which makes it flexibile and extensible to store different kinds of data.

On the other hand, it makes difficult to convert into tabular data because of its nature of semi-structured. For example, in the below XML excerption, the description element can be expanded to multiple lines. The price element can be omitted because it is yet to be determined.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <book id="bk119">  <author>Feng, Jason</author>  <title>Playground</title>  <description>This is the place where Jason puts his fun stuff  mainly related with Python, R and GCP.</description>  </book> |

**Solution**

This is my scribble of the solution.

**Step 1: Read XML files into RDD**

We use spark.read.text to read all the xml files into a DataFrame. The DataFrame is with one column, and the value of each row is the whole content of each xml file. Then we convert it to RDD which we can utilise some low level API to perform the transformation.

|  |  |
| --- | --- |
| 1  2 | *# read each xml file as one row, then convert to RDD*  file\_rdd = spark.read.text("./data/\*.xml", wholetext=True).rdd |

Here is the output of one row in the DataFrame.

|  |  |
| --- | --- |
| 1  2 | [Row(value='<?xml version="1.0"?>\r\n<catalog>\r\n <book id="bk119">\r\n <author>Feng, Jason</author>\r\n <title>Playground</title>\r\n <description>This is the place where Jason puts his fun stuff\r\n mainly related with Python, R and GCP.</description>\r\n </book>\r\n</catalog>')] |

**Step 2: Parse XML files, extract the records, and expand into multiple RDDs**

Now it comes to the key part of the entire process. We need to parse each xml content into records according the pre-defined schema.

First, we define a function using Python standard library [xml.etree.ElementTree](https://docs.python.org/3/library/xml.etree.elementtree.html) to parse and extract the xml elements into a list of records. In this function, we cater for the scenario that some elements are missing which None is returned. It also casts price to float type and publish\_date to date type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | def parse\_xml(rdd):  """  Read the xml string from rdd, parse and extract the elements,  then return a list of list.  """  results = []  root = ET.fromstring(rdd[0])  for b **in** root.findall('book'):  rec = []  rec.append(b.attrib['id'])  for e **in** ELEMENTS\_TO\_EXTRAT:  if b.find(e) **is** None:  rec.append(None)  continue  value = b.find(e).text  if e == 'price':  value = float(value)  elif e == 'publish\_date':  value = datetime.strptime(value, '%Y-%m-%d')  rec.append(value)  results.append(rec)  return results |

Then we use [flatMap](https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.flatMap) function which each input item as the content of an XML file can be mapped to multiple items through the function parse\_xml. flatMap is one of the functions made me “*WoW*” when I first used Spark a few years ago.

|  |  |
| --- | --- |
| 1  2 | *# parse xml tree, extract the records and transform to new RDD*  records\_rdd = file\_rdd.flatMap(parse\_xml) |

**Step 3: Convert RDDs into DataFrame**

We then convert the transformed RDDs to DataFrame with the pre-defined schema.

|  |  |
| --- | --- |
| 1  2 | *# convert RDDs to DataFrame with the pre-defined schema*  book\_df = records\_rdd.toDF(my\_schema) |

The DataFrame looks like below.

|  |  |
| --- | --- |
|  | +-------+--------------------+--------------------+---------------+-----+------------+-------------------+  |book\_id| author| title| genre|price|publish\_date| description|  +-------+--------------------+--------------------+---------------+-----+------------+--------------------+  | bk101|Gambardella, Matthew|XML Developer's G...| Computer|44.95| 2000-10-01|An in-depth look ...|  | bk102| Ralls, Kim| Midnight Rain| Fantasy| 5.95| 2000-12-16|A former architec...|  | bk103| Corets, Eva| Maeve Ascendant| Fantasy| 5.95| 2000-11-17|After the collaps...|  | bk104| Corets, Eva| Oberon's Legacy| Fantasy| 5.95| 2001-03-10|In post-apocalyps...|  | bk105| Corets, Eva| The Sundered Grail| Fantasy| 5.95| 2001-09-10|The two daughters...|  | bk106| Randall, Cynthia| Lover Birds| Romance| 4.95| 2000-09-02|When Carla meets ...| |

**Step 4: Save DataFrame as csv files**

Finally we can save the results as csv files. Spark provides rich set of destination formats, i.e. we can write to JSON, parquet, avro, or even to a table in a database.

|  |  |
| --- | --- |
| 1  2  3 | *# write to csv*  book\_df.write.format("csv").mode("overwrite")\  .save("./output") |

**Conclusion**

This is just one of the showcases of what Spark can help to simplify the data processing especially when dealing with large amount of data. Spark provides both high-level API (DataFrame / DataSet), and low-level API (RDD) which enables us with the flexibility to handle various types of data format. Spark also abstracts the physical parallel computation on the cluster. We just need to focus our codes on the implementation of business logic.

# Read & Write Avro files using Spark

<https://stackoverflow.com/questions/54693110/pyspark-2-4-0-read-avro-from-kafka-with-read-stream-python>

You can include spark-avro package, for example using --packages (adjust versions to match spark installation):

bin/pyspark --packages org.apache.spark:spark-avro\_2.11:2.4.0

and provide your own wrappers:

from pyspark.sql.column import Column, \_to\_java\_column

def from\_avro(col, jsonFormatSchema):

sc = SparkContext.\_active\_spark\_context

avro = sc.\_jvm.org.apache.spark.sql.avro

f = getattr(getattr(avro, "package$"), "MODULE$").from\_avro

return Column(f(\_to\_java\_column(col), jsonFormatSchema))

def to\_avro(col):

sc = SparkContext.\_active\_spark\_context

avro = sc.\_jvm.org.apache.spark.sql.avro

f = getattr(getattr(avro, "package$"), "MODULE$").to\_avro

return Column(f(\_to\_java\_column(col)))

Example usage (adopted from [the official test suite](https://github.com/apache/spark/blob/24e8c27dfe31e6e0a53c89e6ddc36327e537931b/external/avro/src/test/scala/org/apache/spark/sql/avro/AvroFunctionsSuite.scala#L49-L63)):

from pyspark.sql.functions import col, struct

avro\_type\_struct = """

{

"type": "record",

"name": "struct",

"fields": [

{"name": "col1", "type": "long"},

{"name": "col2", "type": "string"}

]

}"""

df = spark.range(10).select(struct(

col("id"),

col("id").cast("string").alias("id2")

).alias("struct"))

avro\_struct\_df = df.select(to\_avro(col("struct")).alias("avro"))

avro\_struct\_df.show(3)

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

| avro|

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

|[00 02 30]|

|[02 02 31]|

|[04 02 32]|

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

only showing top 3 rows

avro\_struct\_df.select(from\_avro("avro", avro\_type\_struct)).show(3)

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

|from\_avro(avro, struct<col1:bigint,col2:string>)|

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

| [0, 0]|

| [1, 1]|

| [2, 2]|

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

only showing top 3 rows

# Hive Tables

<https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html>

* [Specifying storage format for Hive tables](https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html#specifying-storage-format-for-hive-tables)
* [Interacting with Different Versions of Hive Metastore](https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html#interacting-with-different-versions-of-hive-metastore)

Spark SQL also supports reading and writing data stored in [Apache Hive](http://hive.apache.org/). However, since Hive has a large number of dependencies, these dependencies are not included in the default Spark distribution. If Hive dependencies can be found on the classpath, Spark will load them automatically. Note that these Hive dependencies must also be present on all of the worker nodes, as they will need access to the Hive serialization and deserialization libraries (SerDes) in order to access data stored in Hive.

Configuration of Hive is done by placing your hive-site.xml, core-site.xml (for security configuration), and hdfs-site.xml (for HDFS configuration) file in conf/.

When working with Hive, one must instantiate SparkSession with Hive support, including connectivity to a persistent Hive metastore, support for Hive serdes, and Hive user-defined functions. Users who do not have an existing Hive deployment can still enable Hive support. When not configured by the hive-site.xml, the context automatically creates metastore\_db in the current directory and creates a directory configured by spark.sql.warehouse.dir, which defaults to the directory spark-warehouse in the current directory that the Spark application is started. Note that the hive.metastore.warehouse.dir property in hive-site.xml is deprecated since Spark 2.0.0. Instead, use spark.sql.warehouse.dir to specify the default location of database in warehouse. You may need to grant write privilege to the user who starts the Spark application.

**from** **os.path** **import** abspath

**from** **pyspark.sql** **import** SparkSession

**from** **pyspark.sql** **import** Row

*# warehouse\_location points to the default location for managed databases and tables*

warehouse\_location = abspath('spark-warehouse')

spark = SparkSession \

.builder \

.appName("Python Spark SQL Hive integration example") \

.config("spark.sql.warehouse.dir", warehouse\_location) \

.enableHiveSupport() \

.getOrCreate()

*# spark is an existing SparkSession*

spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")

spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

*# Queries are expressed in HiveQL*

spark.sql("SELECT \* FROM src").show()

*# +---+-------+*

*# |key| value|*

*# +---+-------+*

*# |238|val\_238|*

*# | 86| val\_86|*

*# |311|val\_311|*

*# ...*

*# Aggregation queries are also supported.*

spark.sql("SELECT COUNT(\*) FROM src").show()

*# +--------+*

*# |count(1)|*

*# +--------+*

*# | 500 |*

*# +--------+*

*# The results of SQL queries are themselves DataFrames and support all normal functions.*

sqlDF = spark.sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

*# The items in DataFrames are of type Row, which allows you to access each column by ordinal.*

stringsDS = sqlDF.rdd.map(**lambda** row: "Key: %d, Value: %s" % (row.key, row.value))

**for** record **in** stringsDS.collect():

**print**(record)

*# Key: 0, Value: val\_0*

*# Key: 0, Value: val\_0*

*# Key: 0, Value: val\_0*

*# ...*

*# You can also use DataFrames to create temporary views within a SparkSession.*

Record = Row("key", "value")

recordsDF = spark.createDataFrame([Record(i, "val\_" + str(i)) **for** i **in** range(1, 101)])

recordsDF.createOrReplaceTempView("records")

*# Queries can then join DataFrame data with data stored in Hive.*

spark.sql("SELECT \* FROM records r JOIN src s ON r.key = s.key").show()

*# +---+------+---+------+*

*# |key| value|key| value|*

*# +---+------+---+------+*

*# | 2| val\_2| 2| val\_2|*

*# | 4| val\_4| 4| val\_4|*

*# | 5| val\_5| 5| val\_5|*

*# ...*

Find full example code at "examples/src/main/python/sql/hive.py" in the Spark repo.

### Specifying storage format for Hive tables

When you create a Hive table, you need to define how this table should read/write data from/to file system, i.e. the “input format” and “output format”. You also need to define how this table should deserialize the data to rows, or serialize rows to data, i.e. the “serde”. The following options can be used to specify the storage format(“serde”, “input format”, “output format”), e.g. CREATE TABLE src(id int) USING hive OPTIONS(fileFormat 'parquet'). By default, we will read the table files as plain text. Note that, Hive storage handler is not supported yet when creating table, you can create a table using storage handler at Hive side, and use Spark SQL to read it.

|  |  |
| --- | --- |
| **Property Name** | **Meaning** |
| fileFormat | A fileFormat is kind of a package of storage format specifications, including "serde", "input format" and "output format". Currently we support 6 fileFormats: 'sequencefile', 'rcfile', 'orc', 'parquet', 'textfile' and 'avro'. |
| inputFormat, outputFormat | These 2 options specify the name of a corresponding InputFormat and OutputFormat class as a string literal, e.g. org.apache.hadoop.hive.ql.io.orc.OrcInputFormat. These 2 options must be appeared in a pair, and you can not specify them if you already specified the fileFormat option. |
| serde | This option specifies the name of a serde class. When the fileFormat option is specified, do not specify this option if the given fileFormat already include the information of serde. Currently "sequencefile", "textfile" and "rcfile" don't include the serde information and you can use this option with these 3 fileFormats. |
| fieldDelim, escapeDelim, collectionDelim, mapkeyDelim, lineDelim | These options can only be used with "textfile" fileFormat. They define how to read delimited files into rows. |

All other properties defined with OPTIONS will be regarded as Hive serde properties.

### Interacting with Different Versions of Hive Metastore

One of the most important pieces of Spark SQL’s Hive support is interaction with Hive metastore, which enables Spark SQL to access metadata of Hive tables. Starting from Spark 1.4.0, a single binary build of Spark SQL can be used to query different versions of Hive metastores, using the configuration described below. Note that independent of the version of Hive that is being used to talk to the metastore, internally Spark SQL will compile against built-in Hive and use those classes for internal execution (serdes, UDFs, UDAFs, etc).

The following options can be used to configure the version of Hive that is used to retrieve metadata:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Default** | **Meaning** | **Since Version** |
| spark.sql.hive.metastore.version | 2.3.7 | Version of the Hive metastore. Available options are 0.12.0 through 2.3.7 and 3.0.0 through 3.1.2. | 1.4.0 |
| spark.sql.hive.metastore.jars | builtin | Location of the jars that should be used to instantiate the HiveMetastoreClient. This property can be one of four options:   1. builtin   Use Hive 2.3.7, which is bundled with the Spark assembly when -Phive is enabled. When this option is chosen, spark.sql.hive.metastore.version must be either 2.3.7 or not defined.   1. maven   Use Hive jars of specified version downloaded from Maven repositories. This configuration is not generally recommended for production deployments.   1. path   Use Hive jars configured by spark.sql.hive.metastore.jars.path in comma separated format. Support both local or remote paths. The provided jars should be the same version as spark.sql.hive.metastore.version.   1. A classpath in the standard format for the JVM. This classpath must include all of Hive and its dependencies, including the correct version of Hadoop. The provided jars should be the same version as spark.sql.hive.metastore.version. These jars only need to be present on the driver, but if you are running in yarn cluster mode then you must ensure they are packaged with your application. | 1.4.0 |
| spark.sql.hive.metastore.jars.path | (empty) | Comma-separated paths of the jars that used to instantiate the HiveMetastoreClient. This configuration is useful only when spark.sql.hive.metastore.jars is set as path. The paths can be any of the following format:   1. file://path/to/jar/foo.jar 2. hdfs://nameservice/path/to/jar/foo.jar 3. /path/to/jar/(path without URI scheme follow conf fs.defaultFS's URI schema) 4. [http/https/ftp]://path/to/jar/foo.jar   Note that 1, 2, and 3 support wildcard. For example:   1. file://path/to/jar/\*,file://path2/to/jar/\*/\*.jar 2. hdfs://nameservice/path/to/jar/\*,hdfs://nameservice2/path/to/jar/\*/\*.jar | 3.1.0 |
| spark.sql.hive.metastore.sharedPrefixes | com.mysql.jdbc, org.postgresql, com.microsoft.sqlserver, oracle.jdbc | A comma-separated list of class prefixes that should be loaded using the classloader that is shared between Spark SQL and a specific version of Hive. An example of classes that should be shared is JDBC drivers that are needed to talk to the metastore. Other classes that need to be shared are those that interact with classes that are already shared. For example, custom appenders that are used by log4j. | 1.4.0 |
| spark.sql.hive.metastore.barrierPrefixes | (empty) | A comma separated list of class prefixes that should explicitly be reloaded for each version of Hive that Spark SQL is communicating with. For example, Hive UDFs that are declared in a prefix that typically would be shared (i.e. org.apache.spark.\*). | 1.4.0 |

# JDBC To Other Databases

<https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html>

Spark SQL also includes a data source that can read data from other databases using JDBC. This functionality should be preferred over using [JdbcRDD](https://spark.apache.org/docs/latest/api/scala/org/apache/spark/rdd/JdbcRDD.html). This is because the results are returned as a DataFrame and they can easily be processed in Spark SQL or joined with other data sources. The JDBC data source is also easier to use from Java or Python as it does not require the user to provide a ClassTag. (Note that this is different than the Spark SQL JDBC server, which allows other applications to run queries using Spark SQL).

To get started you will need to include the JDBC driver for your particular database on the spark classpath. For example, to connect to postgres from the Spark Shell you would run the following command:

./bin/spark-shell **--driver-class-path** postgresql-9.4.1207.jar **--jars** postgresql-9.4.1207.jar

Tables from the remote database can be loaded as a DataFrame or Spark SQL temporary view using the Data Sources API. Users can specify the JDBC connection properties in the data source options. user and password are normally provided as connection properties for logging into the data sources. In addition to the connection properties, Spark also supports the following case-insensitive options:

|  |  |
| --- | --- |
| **Property Name** | **Meaning** |
| url | The JDBC URL to connect to. The source-specific connection properties may be specified in the URL. e.g., jdbc:postgresql://localhost/test?user=fred&password=secret |
| dbtable | The JDBC table that should be read from or written into. Note that when using it in the read path anything that is valid in a FROM clause of a SQL query can be used. For example, instead of a full table you could also use a subquery in parentheses. It is not allowed to specify dbtable and query options at the same time. |
| query | A query that will be used to read data into Spark. The specified query will be parenthesized and used as a subquery in the FROM clause. Spark will also assign an alias to the subquery clause. As an example, spark will issue a query of the following form to the JDBC Source.  SELECT <columns> FROM (<user\_specified\_query>) spark\_gen\_alias  Below are a couple of restrictions while using this option.   1. It is not allowed to specify dbtable and query options at the same time. 2. It is not allowed to specify query and partitionColumn options at the same time. When specifying partitionColumn option is required, the subquery can be specified using dbtable option instead and partition columns can be qualified using the subquery alias provided as part of dbtable. Example: spark.read.format("jdbc") .option("url", jdbcUrl) .option("query", "select c1, c2 from t1") .load() |
| driver | The class name of the JDBC driver to use to connect to this URL. |
| partitionColumn, lowerBound, upperBound | These options must all be specified if any of them is specified. In addition, numPartitions must be specified. They describe how to partition the table when reading in parallel from multiple workers. partitionColumn must be a numeric, date, or timestamp column from the table in question. Notice that lowerBound and upperBound are just used to decide the partition stride, not for filtering the rows in table. So all rows in the table will be partitioned and returned. This option applies only to reading. |
| numPartitions | The maximum number of partitions that can be used for parallelism in table reading and writing. This also determines the maximum number of concurrent JDBC connections. If the number of partitions to write exceeds this limit, we decrease it to this limit by calling coalesce(numPartitions) before writing. |
| queryTimeout | The number of seconds the driver will wait for a Statement object to execute to the given number of seconds. Zero means there is no limit. In the write path, this option depends on how JDBC drivers implement the API setQueryTimeout, e.g., the h2 JDBC driver checks the timeout of each query instead of an entire JDBC batch. It defaults to 0. |
| fetchsize | The JDBC fetch size, which determines how many rows to fetch per round trip. This can help performance on JDBC drivers which default to low fetch size (e.g. Oracle with 10 rows). This option applies only to reading. |
| batchsize | The JDBC batch size, which determines how many rows to insert per round trip. This can help performance on JDBC drivers. This option applies only to writing. It defaults to 1000. |
| isolationLevel | The transaction isolation level, which applies to current connection. It can be one of NONE, READ\_COMMITTED, READ\_UNCOMMITTED, REPEATABLE\_READ, or SERIALIZABLE, corresponding to standard transaction isolation levels defined by JDBC's Connection object, with default of READ\_UNCOMMITTED. This option applies only to writing. Please refer the documentation in java.sql.Connection. |
| sessionInitStatement | After each database session is opened to the remote DB and before starting to read data, this option executes a custom SQL statement (or a PL/SQL block). Use this to implement session initialization code. Example: option("sessionInitStatement", """BEGIN execute immediate 'alter session set "\_serial\_direct\_read"=true'; END;""") |
| truncate | This is a JDBC writer related option. When SaveMode.Overwrite is enabled, this option causes Spark to truncate an existing table instead of dropping and recreating it. This can be more efficient, and prevents the table metadata (e.g., indices) from being removed. However, it will not work in some cases, such as when the new data has a different schema. It defaults to false. This option applies only to writing. |
| cascadeTruncate | This is a JDBC writer related option. If enabled and supported by the JDBC database (PostgreSQL and Oracle at the moment), this options allows execution of a TRUNCATE TABLE t CASCADE (in the case of PostgreSQL a TRUNCATE TABLE ONLY t CASCADE is executed to prevent inadvertently truncating descendant tables). This will affect other tables, and thus should be used with care. This option applies only to writing. It defaults to the default cascading truncate behaviour of the JDBC database in question, specified in the isCascadeTruncate in each JDBCDialect. |
| createTableOptions | This is a JDBC writer related option. If specified, this option allows setting of database-specific table and partition options when creating a table (e.g., CREATE TABLE t (name string) ENGINE=InnoDB.). This option applies only to writing. |
| createTableColumnTypes | The database column data types to use instead of the defaults, when creating the table. Data type information should be specified in the same format as CREATE TABLE columns syntax (e.g: "name CHAR(64), comments VARCHAR(1024)"). The specified types should be valid spark sql data types. This option applies only to writing. |
| customSchema | The custom schema to use for reading data from JDBC connectors. For example, "id DECIMAL(38, 0), name STRING". You can also specify partial fields, and the others use the default type mapping. For example, "id DECIMAL(38, 0)". The column names should be identical to the corresponding column names of JDBC table. Users can specify the corresponding data types of Spark SQL instead of using the defaults. This option applies only to reading. |
| pushDownPredicate | The option to enable or disable predicate push-down into the JDBC data source. The default value is true, in which case Spark will push down filters to the JDBC data source as much as possible. Otherwise, if set to false, no filter will be pushed down to the JDBC data source and thus all filters will be handled by Spark. Predicate push-down is usually turned off when the predicate filtering is performed faster by Spark than by the JDBC data source. |
| keytab | Location of the kerberos keytab file (which must be pre-uploaded to all nodes either by --files option of spark-submit or manually) for the JDBC client. When path information found then Spark considers the keytab distributed manually, otherwise --files assumed. If both keytab and principal are defined then Spark tries to do kerberos authentication. |
| principal | Specifies kerberos principal name for the JDBC client. If both keytab and principal are defined then Spark tries to do kerberos authentication. |
| refreshKrb5Config | This option controls whether the kerberos configuration is to be refreshed or not for the JDBC client before establishing a new connection. Set to true if you want to refresh the configuration, otherwise set to false. The default value is false. Note that if you set this option to true and try to establish multiple connections, a race condition can occur. One possble situation would be like as follows.   1. refreshKrb5Config flag is set with security context 1 2. A JDBC connection provider is used for the corresponding DBMS 3. The krb5.conf is modified but the JVM not yet realized that it must be reloaded 4. Spark authenticates successfully for security context 1 5. The JVM loads security context 2 from the modified krb5.conf 6. Spark restores the previously saved security context 1 7. The modified krb5.conf content just gone |

*# Note: JDBC loading and saving can be achieved via either the load/save or jdbc methods*

*# Loading data from a JDBC source*

jdbcDF = spark.read \

.format("jdbc") \

.option("url", "jdbc:postgresql:dbserver") \

.option("dbtable", "schema.tablename") \

.option("user", "username") \

.option("password", "password") \

.load()

jdbcDF2 = spark.read \

.jdbc("jdbc:postgresql:dbserver", "schema.tablename",

properties={"user": "username", "password": "password"})

*# Specifying dataframe column data types on read*

jdbcDF3 = spark.read \

.format("jdbc") \

.option("url", "jdbc:postgresql:dbserver") \

.option("dbtable", "schema.tablename") \

.option("user", "username") \

.option("password", "password") \

.option("customSchema", "id DECIMAL(38, 0), name STRING") \

.load()

*# Saving data to a JDBC source*

jdbcDF.write \

.format("jdbc") \

.option("url", "jdbc:postgresql:dbserver") \

.option("dbtable", "schema.tablename") \

.option("user", "username") \

.option("password", "password") \

.save()

jdbcDF2.write \

.jdbc("jdbc:postgresql:dbserver", "schema.tablename",

properties={"user": "username", "password": "password"})

*# Specifying create table column data types on write*

jdbcDF.write \

.option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)") \

.jdbc("jdbc:postgresql:dbserver", "schema.tablename",

properties={"user": "username", "password": "password"})

.

# Spark Streaming with Kafka Example

References

<https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html>

<https://towardsdatascience.com/connecting-the-dots-python-spark-and-kafka-19e6beba6404>

# Regular expression

Regular Expression is one of the powerful tool to wrangle data.Let us see how we can leverage regular expression to extract data.

Let us see the following in sparksql

1. Regex in pyspark
2. Spark regex function
3. Capture and Non Capture groups

(\b(?:[M|m]ar(?:ch)?)\b [0-9]+,?(?: |)\d{4})

* The outer () creates a capturing group
* \b adds boundaries that allow matching instances of words that are not part of another word — note the placement at both the beginning and end which prevents matching on words like **march**ed or counter**march**.
* ?: specifies that the preceding ( is not the start of a capturing group
* [M|m] matches a single character present in the list [] M or| m
* In (?:ch)? the final ? is a quantifier that makes the ch optional
* whitespace matches on whitespace literally
* [0-9] matches any character in the range present in the list
* + is a greedy quantifier that matches all that precedes it between one and unlimited times
* ,? matches on an optional comma
* (?: |) matches the whitespace (or | nothing) literally
* \d{4} matches a 4-digit number

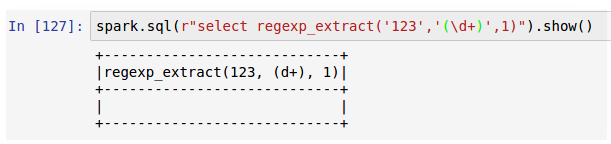
**Regex in pyspark:**

Spark leverage regular expression in the following functions.

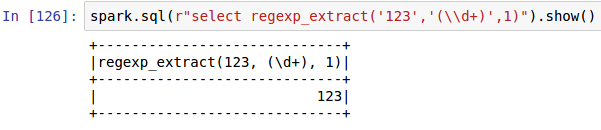
1. Regexp\_extract
2. regexp\_replace
3. rlike

**Escaping Regex expression**

Regex in pyspark internally uses java regex.One of the common issue with regex is escaping backslash as it uses java regex and we will pass raw python string to spark.sql we can see it with a sample example \d represents digit in regex.Let us use spark regexp\_extract to match digit

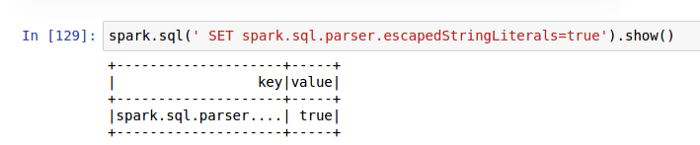


As of now we will assume this function will extract digit but since this string will be converted to java column and backslash have a special meaning in java we need to escape it with another backslash as shown below

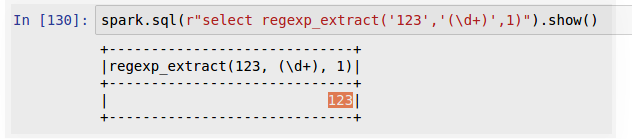


Or we can use the below property to automatically escape \ or similar escape sequences by setting

**SET spark.sql.parser.escapedStringLiterals=true**

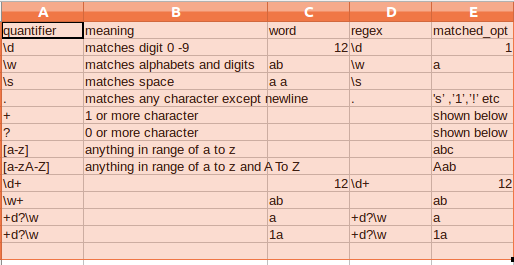


Now we can see the first expression itself gives the below result.



Let us continue the work with default spark setting and we will add backslash

Before going further I will add some meta characters which are building blocks of regex(will explain in detail below)

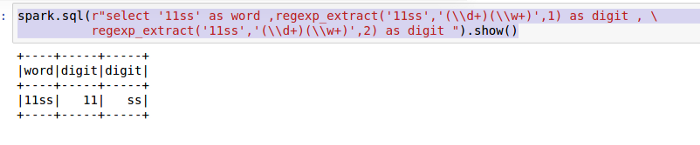


**Regexp\_extract**:

it can be used to extract a part of data by using braces.Regexp\_extract requires 3 arguments

1. data-Column or string from which we want to extract data
2. pattern-regex pattern which we want to extract
3. match group-part of match we need to extract

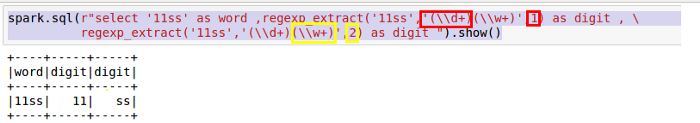
For example in the example below consider we need to extract digit and words seperately and add as 2 diff column from the word ‘11ss’ which can be extracted as shown below



here in regexp\_extract ‘11ss’ represent the column

Since we want to extract number first we are adding \\d and + is added to match more than one number like 11 or 12 and so on and 1 is added as we want to match first group matched as highlighted in red

and then we want to extract the word after digit so we use \w+ to match word after digit and since it is second capture group identified by braces highlighted in yellow



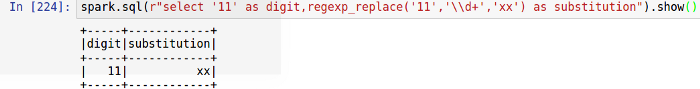
**regexp\_replace:**

It can be used to replace the given pattern with a replacement as the name suggest

It accepts 3 parameters

1. data-column or string
2. pattern -to be replaced
3. Replacement- string with which the pattern to be replaced

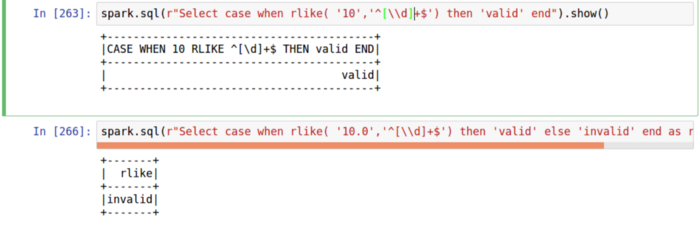
For example to replace digit by space we can use below regex



**Rlike**:

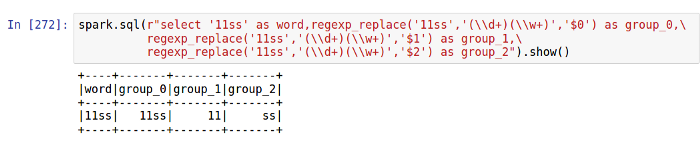
It is use to check if a match is found and can be used with where clauses rather than select clause

For example we want to validate if the amount column contains only integer else we should say as not valid as shown below

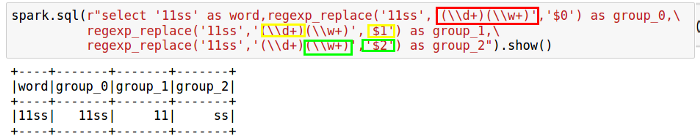


**Capture groups:**

Capture is concept in regex expression where we need to use the captured data in regexp\_replace with replace part.we can access the captured group by using dollar sign($) and it will start from index zero to no of brackets



For example we have used 2 brackets and when we replace with $0(red) it will use the whole group and $1 indicate first bracket(yellow) and $2 can be used for second group(green)



for example a common use case is to mock sensitive data like card with x

so we are going to hide the digits alone from below string for security reason



sssa112ss in this string we will replace digit with x with the above regex

Lot of things in above regex I will try to explain each part

1. pattern part

https://miro.medium.com/max/700/1*hdd4WRi2PlEjVhUe98t0xQ.png

In pattern we have [a-z]+ which will match anything in a to z with one or more repetition like ‘s’ or ss etc (yellow)(Capture group 1)

\d+ matches matches digit (red)(Capture group 2)

[a-z]+ which will match anything in a to z with one or more repetition like ‘s’ or ss etc (green)(Capture group 3)

2.Replace Part-

In replace part we want to replace only the capture group 2 and keep the remaining part .So we will be using capture groups which will capture each groups above and can be accessed by $1x$3 in replace part

**PySpark When Otherwise | SQL Case When Usage**

PySpark When Otherwise and SQL Case When on DataFrame with Examples – Similar to SQL and programming languages, PySpark supports a way to check multiple conditions in sequence and returns a value when the first condition met by using **SQL like case when** and **when().otherwise()** expressions, these works similar to “Switch" and "if then else" statements.

**PySpark When Otherwise** – when() is a SQL function that returns a Column type and otherwise() is a function of Column, if otherwise() is not used, it returns a None/NULL value.

**PySpark SQL Case When** – This is similar to SQL expression, Usage: CASE WHEN cond1 THEN result WHEN cond2 THEN result... ELSE result END.

First, let’s [create a DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/)

from pyspark.sql import SparkSession

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

data = [("James","M",60000),("Michael","M",70000),

("Robert",None,400000),("Maria","F",500000),

("Jen","",None)]

columns = ["name","gender","salary"]

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

df.show()

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

| name|gender|salary|

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

| James| M| 60000|

|Michael| M| 70000|

| Robert| null|400000|

| Maria| F|500000|

| Jen| | null|

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

**1. Using when() otherwise() on PySpark DataFrame.**

PySpark when() is SQL function, in order to use this first you should import and this returns a Column type, otherwise() is a function of Column, when otherwise() not used and none of the conditions met it assigns None (Null) value. Usage would be like when(condition).otherwise(default).

when() function take 2 parameters, first param takes a condition and second takes a literal value or Column, if condition evaluates to true then it returns a value from second param.

The below code snippet replaces the value of gender with a new derived value, when conditions not matched, we are assigning “Unknown” as value, for null assigning empty.

from pyspark.sql.functions import when

df2 = df.withColumn("new\_gender", when(df.gender == "M","Male")

.when(df.gender == "F","Female")

.when(df.gender.isNull() ,"")

.otherwise(df.gender))

df2.show()

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

| name|gender|salary|new\_gender|

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

| James| M| 60000| Male|

|Michael| M| 70000| Male|

| Robert| null|400000| |

| Maria| F|500000| Female|

| Jen| | null| |

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

Using with select()

df2=df.select(col("\*"),when(df.gender == "M","Male")

.when(df.gender == "F","Female")

.when(df.gender.isNull() ,"")

.otherwise(df.gender).alias("new\_gender"))

This yields same output as above.

**2. PySpark SQL Case When on DataFrame.**

If you have a SQL background you might have familiar with [Case When statement](https://www.w3schools.com/sql/sql_case.asp) that is used to execute a sequence of conditions and returns a value when the first condition met, similar to SWITH and IF THEN ELSE statements. Similarly, PySpark SQL Case When statement can be used on DataFrame, below are some of the examples of using with withColumn(), select(), selectExpr() utilizing expr() function.

**Syntax of SQL CASE WHEN ELSE END**

CASE

WHEN condition1 THEN result\_value1

WHEN condition2 THEN result\_value2

-----

-----

ELSE result

END;

* **CASE** is the start of the expression
* Clause **WHEN** takes a condition, if condition true it returns a value from **THEN**
* If the condition is false it goes to the next condition and so on.
* If none of the condition matches, it returns a value from the **ELSE** clause.
* **END** is to end the expression

**2.1 Using Case When Else on DataFrame using withColumn() & select()**

Below example uses [PySpark SQL expr() Function](https://sparkbyexamples.com/pyspark/pyspark-sql-expr-expression-function/) to express SQL like expressions.

from pyspark.sql.functions import expr, col

#Using Case When on withColumn()

df3 = df.withColumn("new\_gender", expr("CASE WHEN gender = 'M' THEN 'Male' " +

"WHEN gender = 'F' THEN 'Female' WHEN gender IS NULL THEN ''" +

"ELSE gender END"))

df3.show(truncate=False)

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

|name |gender|salary|new\_gender|

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

|James |M |60000 |Male |

|Michael|M |70000 |Male |

|Robert |null |400000| |

|Maria |F |500000|Female |

|Jen | |null | |

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

#Using Case When on select()

df4 = df.select(col("\*"), expr("CASE WHEN gender = 'M' THEN 'Male' " +

"WHEN gender = 'F' THEN 'Female' WHEN gender IS NULL THEN ''" +

"ELSE gender END").alias("new\_gender"))

**2.2 Using Case When on SQL Expression**

You can also use Case When with SQL statement after creating a temporary view. This returns a similar output as above.

df.createOrReplaceTempView("EMP")

spark.sql("select name, CASE WHEN gender = 'M' THEN 'Male' " +

"WHEN gender = 'F' THEN 'Female' WHEN gender IS NULL THEN ''" +

"ELSE gender END as new\_gender from EMP").show()

**2.3. Multiple Conditions using & and | operator**

We often need to check with multiple conditions, below is an example of using PySpark When Otherwise with multiple conditions by using and (&) or (|) coperators. To explain this I will use a new set of data to make it simple.

df5.withColumn("new\_column", when(col("code") == "a" | col("code") == "d", "A")

.when(col("code") == "b" & col("amt") == "4", "B")

.otherwise("A1")).show()

Output:

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

| id|code|amt|new\_column|

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

| 66| a| 4| A|

| 67| a| 0| A|

| 70| b| 4| B|

| 71| d| 4| A|

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

**Complete Example – PySpark When Otherwise | SQL Case When**

from pyspark.sql import SparkSession

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

data = [("James","M",60000), ("Michael","M",70000),

("Robert",None,400000), ("Maria","F",500000),

("Jen","",None)]

columns = ["name","gender","salary"]

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

df.show()

#Using When otherwise

from pyspark.sql.functions import when,col

df2 = df.withColumn("new\_gender", when(df.gender == "M","Male")

.when(df.gender == "F","Female")

.when(df.gender.isNull() ,"")

.otherwise(df.gender))

df2.show()

df2=df.select(col("\*"),when(df.gender == "M","Male")

.when(df.gender == "F","Female")

.when(df.gender.isNull() ,"")

.otherwise(df.gender).alias("new\_gender"))

df2.show()

# Using SQL Case When

from pyspark.sql.functions import expr

df3 = df.withColumn("new\_gender", expr("CASE WHEN gender = 'M' THEN 'Male' " +

"WHEN gender = 'F' THEN 'Female' WHEN gender IS NULL THEN ''" +

"ELSE gender END"))

df3.show()

df4 = df.select(col("\*"), expr("CASE WHEN gender = 'M' THEN 'Male' " +

"WHEN gender = 'F' THEN 'Female' WHEN gender IS NULL THEN ''" +

"ELSE gender END").alias("new\_gender"))

df.createOrReplaceTempView("EMP")

spark.sql("select name, CASE WHEN gender = 'M' THEN 'Male' " +

"WHEN gender = 'F' THEN 'Female' WHEN gender IS NULL THEN ''" +

"ELSE gender END as new\_gender from EMP").show()

**Conclusion:**

In this article, you have learned how to use Pyspark SQL “case when” and “when otherwise” on Dataframe by leveraging example like checking with NUll/None, applying with multiple conditions using AND (&), OR (|) logical operators. I hope you like this article.

Happy Learning !!

**References**

* <https://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html>

**PySpark SQL expr() (Expression ) Function**

PySpark expr() is a SQL function to execute SQL-like expressions and to use an existing DataFrame column value as an expression argument to Pyspark built-in functions. Most of the commonly used SQL functions are either part of the [PySpark Column class](https://sparkbyexamples.com/pyspark/pyspark-column-functions/) or built-in pyspark.sql.functions API, besides these PySpark also supports many other SQL functions, so in order to use these, you have to use expr() function.

Below are 2 use cases of **PySpark expr() funcion**.

* First, allowing to use of SQL-like functions that are not present in PySpark Column type & pyspark.sql.functions API. for example CASE WHEN, regr\_count().
* Second, it extends the PySpark SQL Functions by allowing to use DataFrame columns in functions for expression. for example, if you wanted to add a month value from a column to a Date column. I will explain this in the example below.

**1. PySpark expr() Syntax**

Following is syntax of the expr() function.

expr(str)

expr() function takes SQL expression as a string argument, executes the expression, and returns a PySpark Column type. Expressions provided with this function are not a compile-time safety like DataFrame operations.

**2. PySpark SQL expr() Function Examples**

Below are some of the examples of using expr() SQL function.

**2.1 Concatenate Columns using || (similar to SQL)**

If you have SQL background, you pretty much familiar using || to concatenate values from two string columns, you can use expr() expression to do exactly same.

#Concatenate columns using || (sql like)

data=[("James","Bond"),("Scott","Varsa")]

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

df.withColumn("Name",expr(" col1 ||','|| col2")).show()

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

| col1| col2| Name|

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

|James| Bond| James,Bond|

|Scott|Varsa|Scott,Varsa|

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

**2.2 Using SQL CASE WHEN with expr()**

[PySpark doesn’t have SQL Like CASE WHEN](https://sparkbyexamples.com/pyspark/pyspark-when-otherwise/) so in order to use this on [PySpark DataFrame withColumn()](https://sparkbyexamples.com/pyspark/pyspark-withcolumn/) or select(), you should use expr() function with expression as shown below.

Here, I have used CASE WHEN expression on withColumn() by using expr(), this example updates an existing column gender with the derived values, **M for male, F for Female,** and unknown**for others**

from pyspark.sql.functions import expr

data = [("James","M"),("Michael","F"),("Jen","")]

columns = ["name","gender"]

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

#Using CASE WHEN similar to SQL.

from pyspark.sql.functions import expr

df2=df.withColumn("gender", expr("CASE WHEN gender = 'M' THEN 'Male' " +

"WHEN gender = 'F' THEN 'Female' ELSE 'unknown' END"))

df2.show()

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

| name| gender|

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

| James| Male|

|Michael| Female|

| Jen|unknown|

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

If you have any errors in the expression you will get the run time error but not during the compile time.

**2.3 Using an Existing Column Value for Expression**

Most of the PySpark function takes constant literal values but sometimes we need to use a value from an existing column instead of a constant and this is not possible without expr() expression. The below example adds a number of months from an existing column instead of a Python constant.

from pyspark.sql.functions import expr

data=[("2019-01-23",1),("2019-06-24",2),("2019-09-20",3)]

df=spark.createDataFrame(data).toDF("date","increment")

#Add Month value from another column

df.select(df.date,df.increment,

expr("add\_months(date,increment)")

.alias("inc\_date")).show()

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

| date|increment| inc\_date|

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

|2019-01-23| 1|2019-02-23|

|2019-06-24| 2|2019-08-24|

|2019-09-20| 3|2019-12-20|

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

Note that Importing SQL functions are not required when using them with expr(). You see above add\_months() is used without importing.

**2.4 Giving Column Alias along with expr()**

You can also use SQL like syntax to provide the [alias name to the column](https://sparkbyexamples.com/pyspark/pyspark-column-alias-after-groupby/) expression.

# Providing alias using 'as'

from pyspark.sql.functions import expr

df.select(df.date,df.increment,

expr("""add\_months(date,increment) as inc\_date""")

).show()

# This yields same output as above

**2.5 Case Function with expr()**

Below example converts long data type to String type.

# Using Cast() Function

df.select("increment",expr("cast(increment as string) as str\_increment")) \

.printSchema()

root

|-- increment: long (nullable = true)

|-- str\_increment: string (nullable = true)

**2.7 Arithmetic operations**

expr() is also used to provide arithmetic operations, below examples add value 5 to increment and creates a new column new\_increment

# Arthemetic operations

df.select(df.date,df.increment,

expr("increment + 5 as new\_increment")

).show()

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

| date|increment|new\_increment|

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

|2019-01-23| 1| 6|

|2019-06-24| 2| 7|

|2019-09-20| 3| 8|

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

**2.8 Using Filter with expr()**

[Filter the DataFrame rows](https://sparkbyexamples.com/pyspark/pyspark-where-filter/) can done using expr() expression.

#Use expr() to filter the rows

from pyspark.sql.functions import expr

data=[(100,2),(200,3000),(500,500)]

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

df.filter(expr("col1 == col2")).show()

+----+----+

|col1|col2|

+----+----+

| 500| 500|

+----+----+

**3. Complete Example of PySpark SQL expr() Function**

This example is also available at [GitHub PySpark Examples](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-expr.py) Project.

from pyspark.sql import SparkSession

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

from pyspark.sql.functions import expr

#Concatenate columns

data=[("James","Bond"),("Scott","Varsa")]

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

df.withColumn("Name",expr(" col1 ||','|| col2")).show()

#Using CASE WHEN sql expression

data = [("James","M"),("Michael","F"),("Jen","")]

columns = ["name","gender"]

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

df2 = df.withColumn("gender", expr("CASE WHEN gender = 'M' THEN 'Male' " +

"WHEN gender = 'F' THEN 'Female' ELSE 'unknown' END"))

df2.show()

#Add months from a value of another column

data=[("2019-01-23",1),("2019-06-24",2),("2019-09-20",3)]

df=spark.createDataFrame(data).toDF("date","increment")

df.select(df.date,df.increment,

expr("add\_months(date,increment)")

.alias("inc\_date")).show()

# Providing alias using 'as'

df.select(df.date,df.increment,

expr("""add\_months(date,increment) as inc\_date""")

).show()

# Add

df.select(df.date,df.increment,

expr("increment + 5 as new\_increment")

).show()

# Using cast to convert data types

df.select("increment",expr("cast(increment as string) as str\_increment")) \

.printSchema()

#Use expr() to filter the rows

data=[(100,2),(200,3000),(500,500)]

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

df.filter(expr("col1 == col2")).show()

**Conclusion**

PySpark expr() function provides a way to run SQL like expression with DataFrames, here you have learned how to use expression with select(), withColumn() and to filter the DataFrame rows.

**References**

* <https://spark.apache.org/docs/2.3.1/api/python/_modules/pyspark/sql/functions.html>

# PySpark lit() – Add Literal or Constant to DataFrame

PySpark SQL functions [lit()](https://sparkbyexamples.com/pyspark/pyspark-lit-add-literal-constant/#lit)and [typedLit()](https://sparkbyexamples.com/pyspark/pyspark-lit-add-literal-constant/#typedlit) are used to add a new column to DataFrame by assigning a literal or constant value. Both these functions return [Column type](https://sparkbyexamples.com/pyspark/pyspark-column-functions/) as return type.

Both of these are available in PySpark by importing pyspark.sql.functions

First, let’s create a DataFrame.

import pyspark

from pyspark.sql import SparkSession

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

data = [("111",50000),("222",60000),("333",40000)]

columns= ["EmpId","Salary"]

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

## lit() Function to Add Constant Column

PySpark lit() function is used to add constant or literal value as a new column to the DataFrame.

*Creates a [[Column]] of literal value. The passed in object is returned directly if it is already a [[Column]]. If the object is a Scala Symbol, it is converted into a [[Column]] also. Otherwise, a new [[Column]] is created to represent the literal value*

Let’s take a look at some examples.

### Example 1: Simple usage of lit() function

Let’s see an example of how to create a new column with constant value using lit() [Spark SQL function](https://sparkbyexamples.com/spark/spark-sql-functions-understanding/). On the below snippet, we are creating a new column by adding a literal ‘1’ to PySpark DataFrame.

from pyspark.sql.functions import col,lit

df2 = df.select(col("EmpId"),col("Salary"),lit("1").alias("lit\_value1"))

df2.show(truncate=False)

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

|EmpId|Salary|lit\_value1|

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

| 111| 50000| 1|

| 222| 60000| 1|

| 333| 40000| 1|

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

Adding the same constant literal to all records in DataFrame may not be real-time useful so let’s see another example.

### Example 2 : lit() function with withColumn

The following example shows how to use pyspark lit() function using withColumn to derive a new column based on some conditions.

from pyspark.sql.functions import when, lit, col

df3 = df2.withColumn("lit\_value2", when(col("Salary") >=40000 & col("Salary") <= 50000,lit("100")).otherwise(lit("200")))

df3.show(truncate=False)

Below is the output for the above code snippet.

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

|EmpId|Salary|lit\_value1|lit\_value2|

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

| 111| 50000| 1| 100|

| 222| 60000| 1| 200|

| 333| 40000| 1| 100|

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

## typedLit() Function – Syntax

Difference between lit() and typedLit() is that, typedLit function can handle collection types e.g.: Array, Dictionary(map) e.t.c

## Complete Example of How to Add Constant Column

import pyspark

from pyspark.sql import SparkSession

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

data = [("111",50000),("222",60000),("333",40000)]

columns= ["EmpId","Salary"]

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

df.printSchema()

df.show(truncate=False)

from pyspark.sql.functions import col,lit

df2 = df.select(col("EmpId"),col("Salary"),lit("1").alias("lit\_value1"))

df2.show(truncate=False)

from pyspark.sql.functions import when

df3 = df2.withColumn("lit\_value2", when(col("Salary") >=40000 & col("Salary") <= 50000,lit("100")).otherwise(lit("200")))

df3.show(truncate=False)

### Conclusion:

You have learned multiple ways to add a constant literal value to DataFrame using PySpark lit() function and have learned the difference between lit and typedLit functions.

When possible try to use predefined PySpark functions as they are a little bit more compile-time safety and perform better when compared to user-defined functions. If your application is critical on performance try to avoid using custom UDF functions as these are not guarantee on performance.

# PySpark Convert String to Array Column

PySpark SQL provides split() function to convert delimiter separated String to an Array (StringType to ArrayType) column on DataFrame. This can be done by splitting a string column based on a delimiter like space, comma, pipe e.t.c, and converting it into [ArrayType](https://sparkbyexamples.com/spark/spark-array-arraytype-dataframe-column/).

In this article, I will explain converting String to Array column using split() function on DataFrame and SQL query.

## Split() function syntax

PySpark SQL split() is grouped under [Array Functions](https://sparkbyexamples.com/spark/spark-sql-array-functions/) in PySpark [SQL Functions](https://sparkbyexamples.com/spark/spark-sql-functions-understanding/)class with the below syntax.

pyspark.sql.functions.split(str, pattern, limit=-1)

The split() function takes the first argument as the DataFrame column of type String and the second argument string delimiter that you want to split on. You can also use the pattern as a delimiter. This function returns pyspark.sql.Column of type Array.

Before we start with usage, first, let’s create a DataFrame with a string column with text separated with comma delimiter

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.appName('SparkByExamples.com') \

.getOrCreate()

data = [("James, A, Smith","2018","M",3000),

("Michael, Rose, Jones","2010","M",4000),

("Robert,K,Williams","2010","M",4000),

("Maria,Anne,Jones","2005","F",4000),

("Jen,Mary,Brown","2010","",-1)

]

columns=["name","dob\_year","gender","salary"]

df=spark.createDataFrame(data,columns)

df.printSchema()

This yields the below output. As you notice we have a name column with takens firstname, middle and lastname with comma separated.

root

|-- name: string (nullable = true)

|-- dob\_year: string (nullable = true)

|-- gender: string (nullable = true)

|-- salary: integer (nullable = false)

## PySpark Convert String to Array Column

Below PySpark example snippet splits the String column name on comma delimiter and convert it to an Array. If you do not need the original column, use drop() to remove the column.

from pyspark.sql.functions import split, col

df2 = df.select(split(col("name"),",").alias("NameArray")) \

.drop("name")

df2.printSchema()

df2.show()

This yields below output. As you see below schema NameArray is a array type.

root

|-- NameArray: array (nullable = true)

| |-- element: string (containsNull = true)

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

|NameArray |

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

|[James, A, Smith] |

|[Michael, Rose, Jones]|

|[Robert, K, Williams] |

|[Maria, Anne, Jones] |

|[Jen, Mary, Brown] |

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

## Convert String to Array Column using SQL Query

Since PySpark provides a way to execute the raw SQL, let’s learn how to write the same example using Spark SQL expression.

In order to use raw SQL, first, you need to create a table using createOrReplaceTempView(). This creates a temporary view from the Dataframe and this view is available lifetime of the current Spark context.

df.createOrReplaceTempView("PERSON")

spark.sql("select SPLIT(name,',') as NameArray from PERSON") \

.show()

This yields the same output as above example.

## Complete Example

Below is the complete example of splitting an String type column based on a delimiter or patterns and converting into ArrayType column.

This example is also available at [PySpark-Examples GitHub project](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-string-to-array.py) for reference.

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.appName('SparkByExamples.com') \

.getOrCreate()

data = [("James, A, Smith","2018","M",3000),

("Michael, Rose, Jones","2010","M",4000),

("Robert,K,Williams","2010","M",4000),

("Maria,Anne,Jones","2005","F",4000),

("Jen,Mary,Brown","2010","",-1)

]

columns=["name","dob\_year","gender","salary"]

df=spark.createDataFrame(data,columns)

df.printSchema()

df.show(truncate=False)

from pyspark.sql.functions import split, col

df2 = df.select(split(col("name"),",").alias("NameArray")) \

.drop("name")

df2.printSchema()

df2.show()

df.createOrReplaceTempView("PERSON")

spark.sql("select SPLIT(name,',') as NameArray from PERSON") \

.show()

### Conclusion

In this simple article, you have learned how to Convert the string column into an array column by splitting the string by delimiter and also learned how to use the split function on PySpark SQL expression.

# PySpark – Convert array column to a String

In this PySpark article, I will explain how to convert an array of String column on DataFrame to a String column (separated or concatenated with a comma, space, or any delimiter character) using PySpark function concat\_ws() (translates to concat with separator), and with SQL expression using Scala example.

When curating data on DataFrame we may want to convert the Dataframe with complex [struct datatypes](https://sparkbyexamples.com/spark/spark-sql-structtype-on-dataframe/), [arrays](https://sparkbyexamples.com/spark/spark-array-arraytype-dataframe-column/)and maps to a flat structure. here we will see how to convert array type to string type.

Before we start, first let’s [create a DataFrame](https://sparkbyexamples.com/pyspark/different-ways-to-create-dataframe-in-pyspark/) with array of string column.

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

columns = ["name","languagesAtSchool","currentState"]

data = [("James,,Smith",["Java","Scala","C++"],"CA"), \

("Michael,Rose,",["Spark","Java","C++"],"NJ"), \

("Robert,,Williams",["CSharp","VB"],"NV")]

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

df.printSchema()

df.show(truncate=False)

In this example “languagesAtSchool” is a column of type array. In the next section, we will convert this to a String. This example yields below schema and DataFrame.

root

|-- name: string (nullable = true)

|-- languagesAtSchool: array (nullable = true)

| |-- element: string (containsNull = true)

|-- currentState: string (nullable = true)

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

|name |languagesAtSchool |currentState|

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

|James,,Smith |[Java, Scala, C++]|CA |

|Michael,Rose, |[Spark, Java, C++]|NJ |

|Robert,,Williams|[CSharp, VB] |NV |

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

## Convert an array of String to String column using concat\_ws()

In order to convert array to a string, PySpark SQL provides a built-in function concat\_ws() which takes delimiter of your choice as a first argument and array column (type Column) as the second argument.

**Syntax**

concat\_ws(sep, \*cols)

**Usage**

In order to use concat\_ws() function, you need to import it using pyspark.sql.functions.concat\_ws . Since this function takes the Column type as a second argument, you need to use col().

from pyspark.sql.functions import col, concat\_ws

df2 = df.withColumn("languagesAtSchool",

concat\_ws(",",col("languagesAtSchool")))

df2.printSchema()

df2.show(truncate=False)

This yields below output

root

|-- name: string (nullable = true)

|-- languagesAtSchool: string (nullable = false)

|-- currentState: string (nullable = true)

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

|name |languagesAtSchool|currentState|

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

|James,,Smith |Java,Scala,C++ |CA |

|Michael,Rose, |Spark,Java,C++ |NJ |

|Robert,,Williams|CSharp,VB |NV |

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

## Using PySpark SQL expression

You can also use concat\_ws() function with SQL expression.

df.createOrReplaceTempView("ARRAY\_STRING")

spark.sql("select name, concat\_ws(',',languagesAtSchool) as languagesAtSchool," + \

" currentState from ARRAY\_STRING") \

.show(truncate=False)

## Complete Example

Below is a complete PySpark DataFrame example of converting an array of String column to a String using a Scala example.

import pyspark

from pyspark.sql import SparkSession

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

columns = ["name","languagesAtSchool","currentState"]

data = [("James,,Smith",["Java","Scala","C++"],"CA"), \

("Michael,Rose,",["Spark","Java","C++"],"NJ"), \

("Robert,,Williams",["CSharp","VB"],"NV")]

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

df.printSchema()

df.show(truncate=False)

from pyspark.sql.functions import col, concat\_ws

df2 = df.withColumn("languagesAtSchool",

concat\_ws(",",col("languagesAtSchool")))

df2.printSchema()

df2.show(truncate=False)

df.createOrReplaceTempView("ARRAY\_STRING")

spark.sql("select name, concat\_ws(',',languagesAtSchool) as languagesAtSchool," + \

" currentState from ARRAY\_STRING") \

.show(truncate=False)

This example is also available at the [PySpark Github example project](https://github.com/spark-examples/pyspark-examples/blob/master/pyspark-array-string.py) for reference.

# Pyspark – Get substring() from a column

In PySpark, the substring() function is used to extract the substring from a DataFrame string column by providing the position and length of the string you wanted to extract.

In this tutorial, I have explained with an example of getting substring of a column using substring() from pyspark.sql.functions and using substr() from pyspark.sql.Column type.

## Using SQL function substring()

Using the substring() function of pyspark.sql.functions module we can extract a substring or slice of a string from the DataFrame column by providing the position and length of the string you wanted to slice.

substring(str, pos, len)

Note: Please note that the position is not zero based, but 1 based index.

Below is an example of Pyspark substring() using [withColumn()](https://sparkbyexamples.com/pyspark/pyspark-withcolumn/).

data = [(1,"20200828"),(2,"20180525")]

columns=["id","date"]

df=spark.createDataFrame(data,columns)

df.withColumn('year', substring('date', 1,4))\

.withColumn('month', substring('date', 5,2))\

.withColumn('day', substring('date', 7,2))

df.printSchema()

df.show(truncate=False)

In above example, we have created a DataFrame with two columns, id and date. Here date is in the form “year month day”. HereI have used substring() on date column to return sub strings of date as year, month, day respectively. Below is the output.

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

|id |date |year|month|day|

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

|1 |20200828|2020|08 |28 |

|2 |20180525|2018|05 |25 |

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

## 2. Using substring() with select()

In Pyspark we can get substring() of a column using [select](https://sparkbyexamples.com/pyspark/select-columns-from-pyspark-dataframe/). Above example can bed written as below.

df.select('date', substring('date', 1,4).alias('year'), \

substring('date', 5,2).alias('month'), \

substring('date', 7,2).alias('day'))

## 3.Using substring() with selectExpr()

Sample example using **selectExpr**to get sub string of column(date) as year,month,day. Below is the code that gives same output as above.

df.selectExpr('date', 'substring(date, 1,4) as year', \

'substring(date, 5,2) as month', \

'substring(date, 7,2) as day')

## 4. Using substr() from Column type

Below is the example of getting substring using substr() function from pyspark.sql.Column type in Pyspark.

df3=df.withColumn('year', col('date').substr(1, 4))\

.withColumn('month',col('date').substr(5, 2))\

.withColumn('day', col('date').substr(7, 2))

The above example gives output same as the above mentioned examples.

## Complete Example of PySpark substring()

import pyspark

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, substring

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

data = [(1,"20200828"),(2,"20180525")]

columns=["id","date"]

df=spark.createDataFrame(data,columns)

#Using SQL function substring()

df.withColumn('year', substring('date', 1,4))\

.withColumn('month', substring('date', 5,2))\

.withColumn('day', substring('date', 7,2))

df.printSchema()

df.show(truncate=False)

#Using select

df1=df.select('date', substring('date', 1,4).alias('year'), \

substring('date', 5,2).alias('month'), \

substring('date', 7,2).alias('day'))

#Using with selectExpr

df2=df.selectExpr('date', 'substring(date, 1,4) as year', \

'substring(date, 5,2) as month', \

'substring(date, 7,2) as day')

#Using substr from Column type

df3=df.withColumn('year', col('date').substr(1, 4))\

.withColumn('month',col('date').substr(5, 2))\

.withColumn('day', col('date').substr(7, 2))

df3.show()

### Conclusion

In this session, we have learned different ways of getting substring of a column in PySpark DataFarme. I hope you liked it! Keep practicing. And do comment in the comment section for any kind of questions!!

# PySpark Replace Column Values in DataFrame

You can replace column values of PySpark DataFrame by using SQL string functions regexp\_replace(), translate(), and overlay() with Python examples.

In this article, I will cover examples of how to replace part of a string with another string, replace all columns, change values conditionally, replace values from a python dictionary, replace column value from another DataFrame column e.t.c

First, let’s create a PySpark DataFrame with some addresses and will use this DataFrame to explain how to replace column values.

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[1]").appName("SparkByExamples.com").getOrCreate()

address = [(1,"14851 Jeffrey Rd","DE"),

(2,"43421 Margarita St","NY"),

(3,"13111 Siemon Ave","CA")]

df =spark.createDataFrame(address,["id","address","state"])

df.show()

## 1. PySpark Replace String Column Values

By using PySpark SQL function regexp\_replace() you can replace a column value with a string for another string/substring. regexp\_replace() uses **Java regex** for matching, if the regex does not match it returns an empty string, the below example replace the street name Rd value with Road string on address column.

#Replace part of string with another string

from pyspark.sql.functions import regexp\_replace

df.withColumn('address', regexp\_replace('address', 'Rd', 'Road')) \

.show(truncate=False)

#+---+------------------+-----+

#|id |address |state|

#+---+------------------+-----+

#|1 |14851 Jeffrey Road|DE |

#|2 |43421 Margarita St|NY |

#|3 |13111 Siemon Ave |CA |

#+---+------------------+-----+

## 2. Replace Column Values Conditionally

In the above example, we just replaced Rd with Road, but not replaced St and Ave values, let’s see how to replace column values conditionally in PySpark Dataframe by using [when().otherwise() SQL condition function](https://sparkbyexamples.com/pyspark/pyspark-when-otherwise/).

#Replace string column value conditionally

from pyspark.sql.functions import when

df.withColumn('address',

when(df.address.endswith('Rd'),regexp\_replace(df.address,'Rd','Road')) \

.when(df.address.endswith('St'),regexp\_replace(df.address,'St','Street')) \

.when(df.address.endswith('Ave'),regexp\_replace(df.address,'Ave','Avenue')) \

.otherwise(df.address)) \

.show(truncate=False)

#+---+----------------------+-----+

#|id |address |state|

#+---+----------------------+-----+

#|1 |14851 Jeffrey Road |DE |

#|2 |43421 Margarita Street|NY |

#|3 |13111 Siemon Avenue |CA |

#+---+----------------------+-----+

## 3. Replace Column Value with Dictionary (map)

You can also replace column values from the**python dictionary (map)**. In the below example, we replace the string value of the state column with the full abbreviated name from a dictionary **key-value pair**, in order to do so I use [PySpark map() transformation to loop through each row of DataFrame](https://sparkbyexamples.com/pyspark/pyspark-map-transformation/).

#Replace values from Dictionary

stateDic={'CA':'California','NY':'New York','DE':'Delaware'}

df2=df.rdd.map(lambda x:

(x.id,x.address,stateDic[x.state])

).toDF(["id","address","state"])

df2.show()

#+---+------------------+----------+

#| id| address| state|

#+---+------------------+----------+

#| 1| 14851 Jeffrey Rd| Delaware|

#| 2|43421 Margarita St| New York|

#| 3| 13111 Siemon Ave|California|

#+---+------------------+----------+

## 4. Replace Column Value Character by Character

By using translate() string function you can **replace character by character of DataFrame column** value. In the below example, every character of **1** is replaced with **A**, **2** replaced with **B**, and **3** replaced with **C** on the address column.

#Using translate to replace character by character

from pyspark.sql.functions import translate

df.withColumn('address', translate('address', '123', 'ABC')) \

.show(truncate=False)

#+---+------------------+-----+

#|id |address |state|

#+---+------------------+-----+

#|1 |A485A Jeffrey Rd |DE |

#|2 |4C4BA Margarita St|NY |

#|3 |ACAAA Siemon Ave |CA |

#+---+------------------+-----+

## 5. Replace Column with Another Column Value

By using expr() and regexp\_replace() you can **replace column value with a value from another DataFrame column**. In the below example, we match the value from col2 in col1 and replace with col3 to create new\_column. Use [expr() to provide SQL like expressions](https://sparkbyexamples.com/pyspark/pyspark-sql-expr-expression-function/) and is used to refer to another column to perform operations.

#Replace column with another column

from pyspark.sql.functions import expr

df = spark.createDataFrame(

[("ABCDE\_XYZ", "XYZ","FGH")],

("col1", "col2","col3")

)

df.withColumn("new\_column",

expr("regexp\_replace(col1, col2, col3)")

.alias("replaced\_value")

).show()

#+---------+----+----+----------+

#| col1|col2|col3|new\_column|

#+---------+----+----+----------+

#|ABCDE\_XYZ| XYZ| FGH| ABCDE\_FGH|

#+---------+----+----+----------+

## 6. Replace All or Multiple Column Values

If you want to replace values on all or selected DataFrame columns, refer to [How to Replace NULL/None values on all column in PySpark](https://sparkbyexamples.com/pyspark/pyspark-fillna-fill-replace-null-values/) or How to replace [empty string with NULL/None value](https://sparkbyexamples.com/pyspark/pyspark-replace-empty-value-with-none-on-dataframe/)

## 7. Using overlay() Function

Replace column value with a string value from another column.

#Overlay

from pyspark.sql.functions import overlay

df = spark.createDataFrame([("ABCDE\_XYZ", "FGH")], ("col1", "col2"))

df.select(overlay("col1", "col2", 7).alias("overlayed")).show()

#+---------+

#|overlayed|

#+---------+

#|ABCDE\_FGH|

#+---------+

## 8. Complete Example

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[1]").appName("SparkByExamples.com").getOrCreate()

address = [(1,"14851 Jeffrey Rd","DE"),

(2,"43421 Margarita St","NY"),

(3,"13111 Siemon Ave","CA")]

df =spark.createDataFrame(address,["id","address","state"])

df.show()

#Replace string

from pyspark.sql.functions import regexp\_replace

df.withColumn('address', regexp\_replace('address', 'Rd', 'Road')) \

.show(truncate=False)

#Replace string

from pyspark.sql.functions import when

df.withColumn('address',

when(df.address.endswith('Rd'),regexp\_replace(df.address,'Rd','Road')) \

.when(df.address.endswith('St'),regexp\_replace(df.address,'St','Street')) \

.when(df.address.endswith('Ave'),regexp\_replace(df.address,'Ave','Avenue')) \

.otherwise(df.address)) \

.show(truncate=False)

#Replace values from Dictionary

stateDic={'CA':'California','NY':'New York','DE':'Delaware'}

df2=df.rdd.map(lambda x:

(x.id,x.address,stateDic[x.state])

).toDF(["id","address","state"])

df2.show()

#Using translate

from pyspark.sql.functions import translate

df.withColumn('address', translate('address', '123', 'ABC')) \

.show(truncate=False)

#Replace column with another column

from pyspark.sql.functions import expr

df = spark.createDataFrame([("ABCDE\_XYZ", "XYZ","FGH")], ("col1", "col2","col3"))

df.withColumn("new\_column",

expr("regexp\_replace(col1, col2, col3)")

.alias("replaced\_value")

).show()

#Overlay

from pyspark.sql.functions import overlay

df = spark.createDataFrame([("ABCDE\_XYZ", "FGH")], ("col1", "col2"))

df.select(overlay("col1", "col2", 7).alias("overlayed")).show()

### Conclusion

In conclusion regexp\_replace() function is used to replace a string in a DataFrame column with another value, translate() function to replace character by character of column values, overlay() function to overlay string with another column string from start position and number of characters. Finally, you have also learned how to replace column values from a dictionary using Python examples.

Happy Learning !!

## References

* <https://kb.databricks.com/data/null-empty-strings.html>

# PySpark to\_timestamp() – Convert String to Timestamp type

Use <em>to\_timestamp</em>() function to convert String to Timestamp (TimestampType) in PySpark. The converted time would be in a default format of MM-dd-yyyy HH:mm:ss.SSS, I will explain how to use this function with a few examples.

### Syntax – to\_timestamp()

Syntax: to\_timestamp(timestampString:Column)

Syntax: to\_timestamp(timestampString:Column,format:String)

This function has above two signatures that defined in [PySpark SQL Date & Timestamp Functions](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/), the first syntax takes just one argument and the argument should be in Timestamp format ‘MM-dd-yyyy HH:mm:ss.SSS‘, when the format is not in this format, it returns null.

The second signature takes an additional String argument to specify the format of the input Timestamp; this support formats specified in [SimeDateFormat](https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/text/SimpleDateFormat.html). Using this additional argument, you can cast String from any format to Timestamp type in PySpark.

## Convert String to PySpark Timestamp type

In the below example we convert the string pattern which is in PySpark default format to Timestamp type, since the input DataFrame column is in default Timestamp format, we use the first signature for conversion. And the second example uses the cast function to do the same.

from pyspark.sql.functions import \*

df=spark.createDataFrame(

data = [ ("1","2019-06-24 12:01:19.000")],

schema=["id","input\_timestamp"])

df.printSchema()

#Timestamp String to DateType

df.withColumn("timestamp",to\_timestamp("input\_timestamp")) \

.show(truncate=False)

# Using Cast to convert TimestampType to DateType

df.withColumn('timestamp\_string', \

to\_timestamp('timestamp').cast('string')) \

.show(truncate=False)

In this snippet, we just add a new column timestamp by converting the input column from string to Timestamp type.

root

|-- id: string (nullable = true)

|-- timestamp: string (nullable = true)

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

|id |input\_timestamp |timestamp |

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

|1 |2019-06-24 12:01:19.000|2019-06-24 12:01:19|

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

## Custom string format to Timestamp type

This example converts input timestamp string from custom format to PySpark Timestamp type, to do this, we use the second syntax where it takes an additional argument to specify user-defined patterns for date-time formatting,

#when dates are not in Spark TimestampType format 'yyyy-MM-dd HH:mm:ss.SSS'.

#Note that when dates are not in Spark Tiemstamp format, all Spark functions returns null

#Hence, first convert the input dates to Spark DateType using to\_timestamp function

df.select(to\_timestamp(lit('06-24-2019 12:01:19.000'),'MM-dd-yyyy HH:mm:ss.SSSS')) \

.show()

#Displays

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

|to\_timestamp('06-24-2019 12:01:19.000', 'MM-dd-yyyy HH:mm:ss.SSSS')|

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

| 2019-06-24 12:01:19|

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

In case if you want to [convert string to date format](https://sparkbyexamples.com/spark/convert-string-to-date-format-spark-sql/) use to\_date() function. And here is another [example to convert Timestamp to custom string pattern format](https://sparkbyexamples.com/spark/spark-convert-timestamp-to-string/).

## SQL Example

#SQL string to TimestampType

spark.sql("select to\_timestamp('2019-06-24 12:01:19.000') as timestamp")

#SQL CAST timestamp string to TimestampType

spark.sql("select timestamp('2019-06-24 12:01:19.000') as timestamp")

#SQL Custom string to TimestampType

spark.sql("select to\_timestamp('06-24-2019 12:01:19.000','MM-dd-yyyy HH:mm:ss.SSSS') as timestamp")

## Complete Example for quick reference

from pyspark.sql import SparkSession

# Create SparkSession

spark = SparkSession.builder \

.appName('SparkByExamples.com') \

.getOrCreate()

from pyspark.sql.functions import \*

df=spark.createDataFrame(

data = [ ("1","2019-06-24 12:01:19.000")],

schema=["id","input\_timestamp"])

df.printSchema()

#Timestamp String to DateType

df.withColumn("timestamp",to\_timestamp("input\_timestamp")) \

.show(truncate=False)

# Using Cast to convert TimestampType to DateType

df.withColumn('timestamp', \

to\_timestamp('input\_timestamp').cast('string')) \

.show(truncate=False)

df.select(to\_timestamp(lit('06-24-2019 12:01:19.000'),'MM-dd-yyyy HH:mm:ss.SSSS')) \

.show(truncate=False)

#SQL string to TimestampType

spark.sql("select to\_timestamp('2019-06-24 12:01:19.000') as timestamp")

#SQL CAST timestamp string to TimestampType

spark.sql("select timestamp('2019-06-24 12:01:19.000') as timestamp")

#SQL Custom string to TimestampType

spark.sql("select to\_timestamp('06-24-2019 12:01:19.000','MM-dd-yyyy HH:mm:ss.SSSS') as timestamp")

# PySpark to\_date() – Convert Timestamp to Date

PySpark functions provide to\_date() function to convert timestamp to date (DateType), this ideally achieved by just truncating the time part from the Timestamp column. In this tutorial, I will show you a PySpark example of how to convert timestamp to date on DataFrame & SQL.

to\_date() – function formats Timestamp to Date.

Syntax: to\_date(timestamp\_column)

Syntax: to\_date(timestamp\_column,format)

PySpark timestamp (TimestampType) consists of value in the format yyyy-MM-dd HH:mm:ss.SSSS and Date (DateType) format would be yyyy-MM-dd. Use to\_date() function to truncate time from Timestamp or to convert the timestamp to date on DataFrame column.

df=spark.createDataFrame(

data = [ ("1","2019-06-24 12:01:19.000")],

schema=["id","input\_timestamp"])

df.printSchema()

#Displays

root

|-- id: string (nullable = true)

|-- input\_timestamp: string (nullable = true)

## Using to\_date() – Convert Timestamp String to Date

In this example, we will use to\_date() function to convert TimestampType (or string) column to DateType column. The input to this function should be timestamp column or string in TimestampType format and it returns just date in DateType column.

from pyspark.sql.functions import \*

#Timestamp String to DateType

df.withColumn("date\_type",to\_date("input\_timestamp")) \

.show(truncate=False)

#Timestamp Type to DateType

df.withColumn("date\_type",to\_date(current\_timestamp())) \

.show(truncate=False)

#Above Both examples display

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

|id |input\_timestamp |date\_type |

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

|1 |2019-06-24 12:01:19.000|2019-06-24|

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

#Custom Timestamp format to DateType

df.select(to\_date(lit('06-24-2019 12:01:19.000'),'MM-dd-yyyy HH:mm:ss.SSSS')) \

.show()

#Displays

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

|to\_date('06-24-2019 12:01:19.000', 'MM-dd-yyyy HH:mm:ss.SSSS')|

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

| 2019-06-24|

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

## Convert TimestampType (timestamp) to DateType (date)

This example converts the PySpark TimestampType column to DateType.

#Timestamp type to DateType

df.withColumn("ts",to\_timestamp(col("input\_timestamp"))) \

.withColumn("datetype",to\_date(col("ts"))) \

.show(truncate=False)

#Displays

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

|id |input\_timestamp |ts |datetype |

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

|1 |2019-06-24 12:01:19.000|2019-06-24 12:01:19|2019-06-24|

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

## Using Column cast() Function

Here is another way to convert TimestampType (timestamp string) to DateType using cast function.

# Using Cast to convert Timestamp String to DateType

df.withColumn('date\_type', col('input\_timestamp').cast('date')) \

.show(truncate=False)

# Using Cast to convert TimestampType to DateType

df.withColumn('date\_type', to\_timestamp('input\_timestamp').cast('date')) \

.show(truncate=False)

#Displays

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

|id |input\_timestamp |date\_type |

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

|1 |2019-06-24 12:01:19.000|2019-06-24|

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

## PySpark SQL – Convert Timestamp to Date

Following are similar examples using with PySpark SQL. If you are from an SQL background these come in handy.

#SQL TimestampType to DateType

spark.sql("select to\_date(current\_timestamp) as date\_type")

#SQL CAST TimestampType to DateType

spark.sql("select date(to\_timestamp('2019-06-24 12:01:19.000')) as date\_type")

#SQL CAST timestamp string to DateType

spark.sql("select date('2019-06-24 12:01:19.000') as date\_type")

#SQL Timestamp String (default format) to DateType

spark.sql("select to\_date('2019-06-24 12:01:19.000') as date\_type")

#SQL Custom Timeformat to DateType

spark.sql("select to\_date('06-24-2019 12:01:19.000','MM-dd-yyyy HH:mm:ss.SSSS') as date\_type")

## Complete code

from pyspark.sql import SparkSession

# Create SparkSession

spark = SparkSession.builder \

.appName('SparkByExamples.com') \

.getOrCreate()

df=spark.createDataFrame(

data = [ ("1","2019-06-24 12:01:19.000")],

schema=["id","input\_timestamp"])

df.printSchema()

from pyspark.sql.functions import \*

# Using Cast to convert Timestamp String to DateType

df.withColumn('date\_type', col('input\_timestamp').cast('date')) \

.show(truncate=False)

# Using Cast to convert TimestampType to DateType

df.withColumn('date\_type', to\_timestamp('input\_timestamp').cast('date')) \

.show(truncate=False)

df.select(to\_date(lit('06-24-2019 12:01:19.000'),'MM-dd-yyyy HH:mm:ss.SSSS')) \

.show()

#Timestamp String to DateType

df.withColumn("date\_type",to\_date("input\_timestamp")) \

.show(truncate=False)

#Timestamp Type to DateType

df.withColumn("date\_type",to\_date(current\_timestamp())) \

.show(truncate=False)

df.withColumn("ts",to\_timestamp(col("input\_timestamp"))) \

.withColumn("datetype",to\_date(col("ts"))) \

.show(truncate=False)

#SQL TimestampType to DateType

spark.sql("select to\_date(current\_timestamp) as date\_type")

#SQL CAST TimestampType to DateType

spark.sql("select date(to\_timestamp('2019-06-24 12:01:19.000')) as date\_type")

#SQL CAST timestamp string to DateType

spark.sql("select date('2019-06-24 12:01:19.000') as date\_type")

#SQL Timestamp String (default format) to DateType

spark.sql("select to\_date('2019-06-24 12:01:19.000') as date\_type")

#SQL Custom Timeformat to DateType

spark.sql("select to\_date('06-24-2019 12:01:19.000','MM-dd-yyyy HH:mm:ss.SSSS') as date\_type")

# PySpark date\_format() – Convert Date to String format

In PySpark use date\_format() function to convert the DataFrame column from Date to String format. In this tutorial, we will show you a Spark SQL example of how to convert Date to String format using  date\_format() function on DataFrame.

date\_format() – function formats Date to String format. This function supports all Java Date formats specified in [DateTimeFormatter](https://docs.oracle.com/javase/10/docs/api/java/time/format/DateTimeFormatter.html).

**Following are Syntax and Example of date\_format() Function**:

Syntax: date\_format(column,format)

Example: date\_format(current\_timestamp(),"yyyy MM dd").alias("date\_format")

The below code snippet takes the current system date from current\_date() and timestamp from the current\_timestamp() function and converts it to String format on DataFrame.

from pyspark.sql.functions import \*

df=spark.createDataFrame([["1"]],["id"])

df.select(current\_date().alias("current\_date"), \

date\_format(current\_timestamp(),"yyyy MM dd").alias("yyyy MM dd"), \

date\_format(current\_timestamp(),"MM/dd/yyyy hh:mm").alias("MM/dd/yyyy"), \

date\_format(current\_timestamp(),"yyyy MMM dd").alias("yyyy MMMM dd"), \

date\_format(current\_timestamp(),"yyyy MMMM dd E").alias("yyyy MMMM dd E") \

).show()

Output:

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

|current\_date|yyyy MM dd| MM/dd/yyyy|yyyy MMMM dd| yyyy MMMM dd E|

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

| 2021-02-23|2021 02 23|02/23/2021 02:18| 2021 Feb 23|2021 February 23 Tue|

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

Alternatively, you can convert Data to String with SQL by using same functions.

#SQL

spark.sql("select current\_date() as current\_date, "+

"date\_format(current\_timestamp(),'yyyy MM dd') as yyyy\_MM\_dd, "+

"date\_format(current\_timestamp(),'MM/dd/yyyy hh:mm') as MM\_dd\_yyyy, "+

"date\_format(current\_timestamp(),'yyyy MMM dd') as yyyy\_MMMM\_dd, "+

"date\_format(current\_timestamp(),'yyyy MMMM dd E') as yyyy\_MMMM\_dd\_E").show()

## Complete Example of Convert Date to String

from pyspark.sql import SparkSession

# Create SparkSession

spark = SparkSession.builder \

.appName('SparkByExamples.com') \

.getOrCreate()

from pyspark.sql.functions import \*

df=spark.createDataFrame([["1"]],["id"])

df.select(current\_date().alias("current\_date"), \

date\_format(current\_date(),"yyyy MM dd").alias("yyyy MM dd"), \

date\_format(current\_timestamp(),"MM/dd/yyyy hh:mm").alias("MM/dd/yyyy"), \

date\_format(current\_timestamp(),"yyyy MMM dd").alias("yyyy MMMM dd"), \

date\_format(current\_timestamp(),"yyyy MMMM dd E").alias("yyyy MMMM dd E") \

).show()

#SQL

spark.sql("select current\_date() as current\_date, "+

"date\_format(current\_timestamp(),'yyyy MM dd') as yyyy\_MM\_dd, "+

"date\_format(current\_timestamp(),'MM/dd/yyyy hh:mm') as MM\_dd\_yyyy, "+

"date\_format(current\_timestamp(),'yyyy MMM dd') as yyyy\_MMMM\_dd, "+

"date\_format(current\_timestamp(),'yyyy MMMM dd E') as yyyy\_MMMM\_dd\_E").show()

### Conclusion:

In this article, you have learned how to convert Date to String format using the Date function date\_format().

### Related Articles:

* [PySpark – How to Get Current Date & Timestamp](https://sparkbyexamples.com/pyspark/pyspark-current-date-timestamp/)
* [PySpark Convert String to Date Format](https://sparkbyexamples.com/pyspark/pyspark-to_date-convert-string-to-date-format/)
* [PySpark SQL Date and Timestamp Functions](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/)

# PySpark – Difference between two dates (days, months, years)

Using PySpark SQL functions datediff(), months\_between() you can calculate the difference between two dates in days, months, and year, let’s see this by using a DataFrame example. You can also use these to calculate age.

## datediff() Function

First Let’s see getting the difference between two dates using datediff() PySpark function.

from pyspark.sql.functions import \*

data = [("1","2019-07-01"),("2","2019-06-24"),("3","2019-08-24")]

df=spark.createDataFrame(data=data,schema=["id","date"])

df.select(

col("date"),

current\_date().alias("current\_date"),

datediff(current\_date(),col("date")).alias("datediff")

).show()

Output:

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

| date|current\_date|datediff|

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

|2019-07-01| 2021-02-26| 606|

|2019-06-24| 2021-02-26| 613|

|2019-08-24| 2021-02-26| 552|

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

## months\_between() Function

Now, Let’s see how to get month and year differences between two dates using months\_between() function.

from pyspark.sql.functions import \*

df.withColumn("datesDiff", datediff(current\_date(),col("date"))) \

.withColumn("montsDiff", months\_between(current\_date(),col("date"))) \

.withColumn("montsDiff\_round",round(months\_between(current\_date(),col("date")),2)) \

.withColumn("yearsDiff",months\_between(current\_date(),col("date"))/lit(12)) \

.withColumn("yearsDiff\_round",round(months\_between(current\_date(),col("date"))/lit(12),2)) \

.show()

Yields below output. Note that here we use round() function and lit() functions on top of months\_between() to get the year between two dates.

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

| id| date|datesDiff| montsDiff|montsDiff\_round| yearsDiff|yearsDiff\_round|

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

| 1|2019-07-01| 606|19.80645161| 19.81|1.6505376341666667| 1.65|

| 2|2019-06-24| 613|20.06451613| 20.06|1.6720430108333335| 1.67|

| 3|2019-08-24| 552|18.06451613| 18.06|1.5053763441666668| 1.51|

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

Let’s see another example of the difference between two dates when dates are not in PySpark DateType format yyyy-MM-dd. when dates are not in DateType format, all date functions return null. Hence, you need to first convert the input date to Spark DateType using to\_date() function.

from pyspark.sql.functions import \*

data2 = [("1","07-01-2019"),("2","06-24-2019"),("3","08-24-2019")]

df2=spark.createDataFrame(data=data2,schema=["id","date"])

df2.select(

to\_date(col("date"),"MM-dd-yyyy").alias("date"),

current\_date().alias("endDate")

)

## SQL Example

Let’s see how to calculate the difference between two dates in years using PySpark SQL example. similarly you can calculate the days and months between two dates.

spark.sql("select round(months\_between('2019-07-01',current\_date())/12,2) as years\_diff").show()

#### Complete Code:

from pyspark.sql import SparkSession

# Create SparkSession

spark = SparkSession.builder \

.appName('SparkByExamples.com') \

.getOrCreate()

data = [("1","2019-07-01"),("2","2019-06-24"),("3","2019-08-24")]

df=spark.createDataFrame(data=data,schema=["id","date"])

from pyspark.sql.functions import \*

df.select(

col("date"),

current\_date().alias("current\_date"),

datediff(current\_date(),col("date")).alias("datediff")

).show()

df.withColumn("datesDiff", datediff(current\_date(),col("date"))) \

.withColumn("montsDiff", months\_between(current\_date(),col("date"))) \

.withColumn("montsDiff\_round",round(months\_between(current\_date(),col("date")),2)) \

.withColumn("yearsDiff",months\_between(current\_date(),col("date"))/lit(12)) \

.withColumn("yearsDiff\_round",round(months\_between(current\_date(),col("date"))/lit(12),2)) \

.show()

data2 = [("1","07-01-2019"),("2","06-24-2019"),("3","08-24-2019")]

df2=spark.createDataFrame(data=data2,schema=["id","date"])

df2.select(

to\_date(col("date"),"MM-dd-yyyy").alias("date"),

current\_date().alias("endDate")

)

spark.sql("select round(months\_between('2019-07-01',current\_date())/12,2) as years\_diff").show()

### Conclusion:

In this tutorial, you have learned how to calculate days, months, and years between two dates using [PySpark Date and Time functions](https://sparkbyexamples.com/pyspark/pyspark-sql-date-and-timestamp-functions/) datediff(), months\_between(). You can find more information about these functions at the [following blog](https://databricks.com/blog/2015/09/16/apache-spark-1-5-dataframe-api-highlights.html)

Happy Learning !!

# PySpark Convert DataFrame Columns to MapType (Dict)

Problem: How to convert selected or all DataFrame columns to MapType similar to Python Dictionary (Dict) object

Solution: PySpark SQL function create\_map() is used to convert selected DataFrame columns to MapType, create\_map() takes a list of columns you wanted to convert as an argument and returns a MapType column.

Let’s create a DataFrame

from pyspark.sql import SparkSession

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

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

data = [ ("36636","Finance",3000,"USA"),

("40288","Finance",5000,"IND"),

("42114","Sales",3900,"USA"),

("39192","Marketing",2500,"CAN"),

("34534","Sales",6500,"USA") ]

schema = StructType([

StructField('id', StringType(), True),

StructField('dept', StringType(), True),

StructField('salary', IntegerType(), True),

StructField('location', StringType(), True)

])

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

df.printSchema()

df.show(truncate=False)

This yields below output

root

|-- id: string (nullable = true)

|-- dept: string (nullable = true)

|-- salary: integer (nullable = true)

|-- location: string (nullable = true)

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

|id |dept |salary|location|

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

|36636|Finance |3000 |USA |

|40288|Finance |5000 |IND |

|42114|Sales |3900 |USA |

|39192|Marketing|2500 |CAN |

|34534|Sales |6500 |USA |

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

## Convert DataFrame Columns to MapType

Now, using create\_map() SQL function let’s convert PySpark DataFrame columns salary and location to MapType.

#Convert columns to Map

from pyspark.sql.functions import col,lit,create\_map

df = df.withColumn("propertiesMap",create\_map(

lit("salary"),col("salary"),

lit("location"),col("location")

)).drop("salary","location")

df.printSchema()

df.show(truncate=False)

This yields below output.

root

|-- id: string (nullable = true)

|-- dept: string (nullable = true)

|-- propertiesMap: map (nullable = false)

| |-- key: string

| |-- value: string (valueContainsNull = true)

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

|id |dept |propertiesMap |

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

|36636|Finance |[salary -> 3000, location -> USA]|

|40288|Finance |[salary -> 5000, location -> IND]|

|42114|Sales |[salary -> 3900, location -> USA]|

|39192|Marketing|[salary -> 2500, location -> CAN]|

|34534|Sales |[salary -> 6500, location -> USA]|

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

# PySpark Count Distinct from DataFrame

In PySpark, you can use distinct().count() of DataFrame or countDistinct() SQL function to get the count distinct.

distinct() eliminates duplicate records(matching all columns of a Row) from DataFrame, count() returns the count of records on DataFrame. By chaining these you can get the count distinct of PySpark DataFrame.

countDistinct() is a SQL function that could be used to get the count distinct of the selected columns.

Let’s see these two ways with examples.

Before we start, first let’s [create a DataFrame](https://sparkbyexamples.com/spark/different-ways-to-create-a-spark-dataframe/) with some duplicate rows and duplicate values in a column.

sfrom pyspark.sql import SparkSession

spark = SparkSession.builder \

.appName('SparkByExamples.com') \

.getOrCreate()

data = [("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)

]

columns = ["Name","Dept","Salary"]

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

df.show()

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|

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

## Using DataFrame distinct() and count()

On the above DataFrame, we have a total of 10 rows and one row with all values duplicated, performing distinct count ( distinct().count() ) on this DataFrame should get us 9.

print("Distinct Count: " + str(df.distinct().count()))

This yields output **“Distinct Count: 9”**

## Using countDistinct() SQL Function

DataFrame distinct() returns a new DataFrame after eliminating duplicate rows (distinct on all columns). if you want to get count distinct on selected columns, use the PySpark SQL function countDistinct(). This function returns the number of distinct elements in a group.

In order to use this function, you need to import it first.

from pyspark.sql.functions import countDistinct

df2=df.select(countDistinct("department", "salary"))

df2.show()

Yields below output

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

|count(DISTINCT department, salary)|

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

|8 |

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

Note that countDistinct() function returns a value in a Column type hence, you need to collect it to get the value from the DataFrame. And this function can be used to get the distinct count of any number of selected or all columns.

print("Distinct Count of Department & Salary: "+ str(df2.collect()[0][0]))

This outputs **“Distinct Count of Department & Salary: 8”**

## Using SQL to get Count Distinct

df.createOrReplaceTempView("EMP")

spark.sql("select distinct(count(\*)) from EMP").show()

# Displays this on console

+--------+

|count(1)|

+--------+

| 10|

+--------+

## Source Code of PySpark Count Distinct Example

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.appName('SparkByExamples.com') \

.getOrCreate()

data = [("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)

]

columns = ["Name","Dept","Salary"]

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

df.distinct().show()

print("Distinct Count: " + str(df.distinct().count()))

# Using countDistrinct()

from pyspark.sql.functions import countDistinct

df2=df.select(countDistinct("Dept","Salary"))

df2.show()

print("Distinct Count of Department & Salary: "+ str(df2.collect()[0][0]))

## Conclusion

In this article, you have learned how to get count distinct of all columns or selected columns on PySpark DataFrame.

# PySpark lit() – Add Literal or Constant to DataFrame

PySpark SQL functions [lit()](https://sparkbyexamples.com/pyspark/pyspark-lit-add-literal-constant/#lit)and [typedLit()](https://sparkbyexamples.com/pyspark/pyspark-lit-add-literal-constant/#typedlit) are used to add a new column to DataFrame by assigning a literal or constant value. Both these functions return [Column type](https://sparkbyexamples.com/pyspark/pyspark-column-functions/) as return type.

Both of these are available in PySpark by importing pyspark.sql.functions

First, let’s create a DataFrame.

import pyspark

from pyspark.sql import SparkSession

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

data = [("111",50000),("222",60000),("333",40000)]

columns= ["EmpId","Salary"]

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

## lit() Function to Add Constant Column

PySpark lit() function is used to add constant or literal value as a new column to the DataFrame.

*Creates a [[Column]] of literal value. The passed in object is returned directly if it is already a [[Column]]. If the object is a Scala Symbol, it is converted into a [[Column]] also. Otherwise, a new [[Column]] is created to represent the literal value*

Let’s take a look at some examples.

### Example 1: Simple usage of lit() function

Let’s see an example of how to create a new column with constant value using lit() [Spark SQL function](https://sparkbyexamples.com/spark/spark-sql-functions-understanding/). On the below snippet, we are creating a new column by adding a literal ‘1’ to PySpark DataFrame.

from pyspark.sql.functions import col,lit

df2 = df.select(col("EmpId"),col("Salary"),lit("1").alias("lit\_value1"))

df2.show(truncate=False)

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

|EmpId|Salary|lit\_value1|

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

| 111| 50000| 1|

| 222| 60000| 1|

| 333| 40000| 1|

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

Adding the same constant literal to all records in DataFrame may not be real-time useful so let’s see another example.

### Example 2 : lit() function with withColumn

The following example shows how to use pyspark lit() function using withColumn to derive a new column based on some conditions.

from pyspark.sql.functions import when, lit, col

df3 = df2.withColumn("lit\_value2", when(col("Salary") >=40000 & col("Salary") <= 50000,lit("100")).otherwise(lit("200")))

df3.show(truncate=False)

Below is the output for the above code snippet.

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

|EmpId|Salary|lit\_value1|lit\_value2|

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

| 111| 50000| 1| 100|

| 222| 60000| 1| 200|

| 333| 40000| 1| 100|

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

## typedLit() Function – Syntax

Difference between lit() and typedLit() is that, typedLit function can handle collection types e.g.: Array, Dictionary(map) e.t.c

## Complete Example of How to Add Constant Column

import pyspark

from pyspark.sql import SparkSession

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

data = [("111",50000),("222",60000),("333",40000)]

columns= ["EmpId","Salary"]

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

df.printSchema()

df.show(truncate=False)

from pyspark.sql.functions import col,lit

df2 = df.select(col("EmpId"),col("Salary"),lit("1").alias("lit\_value1"))

df2.show(truncate=False)

from pyspark.sql.functions import when

df3 = df2.withColumn("lit\_value2", when(col("Salary") >=40000 & col("Salary") <= 50000,lit("100")).otherwise(lit("200")))

df3.show(truncate=False)

### Conclusion:

You have learned multiple ways to add a constant literal value to DataFrame using PySpark lit() function and have learned the difference between lit and typedLit functions.

When possible try to use predefined PySpark functions as they are a little bit more compile-time safety and perform better when compared to user-defined functions. If your application is critical on performance try to avoid using custom UDF functions as these are not guarantee on performance.