[**Building Spark based Applications using Python** 5](#_Toc19611379)

[Quick revision of Python 3 7](#_Toc19611380)

[Basics of Programming 7](#_Toc19611381)

[Overview of Functions 8](#_Toc19611382)

[Lambda Functions 9](#_Toc19611383)

[Basic File I/O 10](#_Toc19611384)

[Collections and Map Reduce APIs 11](#_Toc19611385)

[Overview of Pandas Data Frames 14](#_Toc19611386)

[**Apache Spark – Getting Started** 15](#_Toc19611387)

[**Setup Spark Environment** 15](#_Toc19611388)

[**Using ITVersity labs** 16](#_Toc19611389)

[**Spark Official Documentation** 17](#_Toc19611390)

[**Quick Review Of APIs** 17](#_Toc19611391)

[**Spark Modules** 18](#_Toc19611392)

[**Spark Data Structures** 19](#_Toc19611393)

[**Simple Application** 20](#_Toc19611394)

[**Spark Framework** 21](#_Toc19611395)

[**Directed Acyclic Graph and Lazy Evaluation** 22](#_Toc19611396)

[RDD, Data Frame, DAG and Lazy Evaluation 23](#_Toc19611397)

[Difference between Python list and RDD 24](#_Toc19611398)

[Resilient Distributed Datasets 24](#_Toc19611399)

[Data Frames 25](#_Toc19611400)

[Overview of Transformations and Actions 26](#_Toc19611401)

[Directed Acyclic Graph and Lazy Evaluation 26](#_Toc19611402)

[Transformations and Actions 27](#_Toc19611403)

[Data Processing Life Cycle 28](#_Toc19611404)

[Sum Of Even Numbers 30](#_Toc19611405)

[Develop Word Count Program 30](#_Toc19611406)

[Get Daily Revenue 32](#_Toc19611407)

[Reading the Data 34](#_Toc19611408)

[Apply Filtering 34](#_Toc19611409)

[Joining Data Sets 35](#_Toc19611410)

[Aggregations 37](#_Toc19611411)

[Sorting Data 39](#_Toc19611412)

[Saving and Conclusion 41](#_Toc19611413)

[Transformations and Actions – Continued 42](#_Toc19611414)

[Overview of Shuffling 43](#_Toc19611415)

[Using aggregateByKey 45](#_Toc19611416)

[mapPartitions 46](#_Toc19611417)

[Global Ranking 47](#_Toc19611418)

[Ranking using groupByKey 47](#_Toc19611419)

[repartition and coalesce 48](#_Toc19611420)

[Development and Deployment Life Cycle 49](#_Toc19611421)

[Development Life Cycle 50](#_Toc19611422)

[Externalize Properties 50](#_Toc19611423)

[Deployment Life Cycle 50](#_Toc19611424)

[Daily Revenue – Validate Locally 51](#_Toc19611425)

[Daily Revenue – Spark Configuration 51](#_Toc19611426)

[Word Count – Spark Configuration 51](#_Toc19611427)

[Accumulators and Broadcast Variables 51](#_Toc19611428)

[Problem Statement 52](#_Toc19611429)

[High-level design 53](#_Toc19611430)

[Validate Paths 54](#_Toc19611431)

[Broadcast Variables 56](#_Toc19611432)

[Using broadcast 57](#_Toc19611433)

[Creating Data Frames and Pre-Defined functions 61](#_Toc19611434)

[Data Frames – Overview 62](#_Toc19611435)

[Reading text data from files 63](#_Toc19611436)

[Reading data from hive 65](#_Toc19611437)

[Reading data from MySQL over JDBC 66](#_Toc19611438)

[Data Frame Operations – Overview 68](#_Toc19611439)

[Spark SQL – Overview 68](#_Toc19611440)

[Functions to manipulate data 68](#_Toc19611441)

[Data Frame Operations – Basic Transformations 69](#_Toc19611442)

[Data Frame Operations – APIs 70](#_Toc19611443)

[Problem Statement – Get daily product revenue 70](#_Toc19611444)

[Projecting or Selecting Data 73](#_Toc19611445)

[Filtering data using where or filter 74](#_Toc19611446)

[Joining Data Sets 75](#_Toc19611447)

[Grouping data and performing aggregations 77](#_Toc19611448)

[Sorting data 79](#_Toc19611449)

[Development Life Cycle 80](#_Toc19611450)

[Exercises 83](#_Toc19611451)

[Data Frame Operations – Analytics or Windowing Functions 83](#_Toc19611452)

[Window Functions – APIs 84](#_Toc19611453)

[Problem Statement 85](#_Toc19611454)

[Creating Window Spec 87](#_Toc19611455)

[Performing aggregations 88](#_Toc19611456)

[Using Windowing Functions 90](#_Toc19611457)

[Ranking Functions 93](#_Toc19611458)

[Development Life Cycle 95](#_Toc19611459)

[Spark SQL – Basic Transformations such as filtering, aggregations, joins etc 98](#_Toc19611460)

[Spark SQL – Overview 99](#_Toc19611461)

[Problem Statement – Get daily product revenue 99](#_Toc19611462)

[Relationship with Hive 102](#_Toc19611463)

[Selection or Projection – select clause 103](#_Toc19611464)

[Filtering data – where clause 103](#_Toc19611465)

[Joining Data Sets 104](#_Toc19611466)

[Aggregations using group by and functions 106](#_Toc19611467)

[Sorting data 107](#_Toc19611468)

[Spark SQL – Analytics Functions or Windowing Functions 109](#_Toc19611469)

[Development Life Cycle (daily product revenue) 109](#_Toc19611470)

[Aggregations, Ranking and Windowing Functions 112](#_Toc19611471)

[Problem Statement – Get top n products per day 114](#_Toc19611472)

[Understanding over, partition by and order by clauses 114](#_Toc19611473)

[Performing aggregations 114](#_Toc19611474)

[Using windowing functions 115](#_Toc19611475)

[Ranking with in each partition or group 115](#_Toc19611476)

[Development Life Cycle 116](#_Toc19611477)

[Different file formats and dealing with custom delimiters 119](#_Toc19611478)

[Overview of write APIs – dataframe.write 119](#_Toc19611479)

[Overview of read APIs – spark.read 121](#_Toc19611480)

[Supported file formats 123](#_Toc19611481)

[Processing text data with custom delimiters 127](#_Toc19611482)

[Persisting or Caching Data Frames 128](#_Toc19611483)

[Compression Concepts and Algorithms 129](#_Toc19611484)

[Compression Algorithms 130](#_Toc19611485)

[Compression – Reading and Writing 133](#_Toc19611486)

[Criteria and Tips 136](#_Toc19611487)

**Building Spark based Applications using Python**

posted on FEBRUARY 1, 2019

<http://wip.itversity.com/lessons/building-spark-based-applications-using-python/>

As part of this lesson we will see how to build scaleable applications using Spark APIs with Python as programming language. We will start with quick review of Python and then we will get into details.

* Quick revision of Python
* Spark Architecture and Execution Modes
* RDD, DAG and Lazy Evaluation
* Basic Transformations and Actions
* Advanced Transformations
* Execution Life Cycle
* Accumulators and Broadcast Variables
* Creating Data Frames and Pre Defined Functions
* Data Frame Operations – Basic Transformations such as filtering, aggregations, joins etc
* Data Frame Operations – Analytics Functions or Windowing Functions
* Spark SQL – Basic Transformations such as filtering, aggregations, joins etc
* Spark SQL – Analytics Functions or Windowing Functions
* Different file formats – text, json, orc, parquet, avrò etc
* Reading text data with custom delimiters
* Compression concepts and algorithms

Lesson Topics

* [Quick revision of Python 3](http://wip.itversity.com/topic/quick-revision-of-python-3/)
* [Apache Spark - Getting Started](http://wip.itversity.com/topic/apache-spark-getting-started/)
* [RDD, Data Frame, DAG and Lazy Evaluation](http://wip.itversity.com/topic/rdd-data-frame-dag-and-lazy-evaluation/)
* [Transformations and Actions](http://wip.itversity.com/topic/transformations-and-actions/)
* [Transformations and Actions - Continued](http://wip.itversity.com/topic/transformations-continued/)
* [Development and Deployment Life Cycle](http://wip.itversity.com/topic/development-and-deployment-life-cycle/)
* [Accumulators and Broadcast Variables](http://wip.itversity.com/topic/accumulators-and-broadcast-variables/)
* [Creating Data Frames and Pre-Defined functions](http://wip.itversity.com/topic/creating-data-frames-and-pre-defined-functions/)
* [Data Frame Operations - Basic Transformations](http://wip.itversity.com/topic/data-frame-operations-basic-transformations-python/)
* [Data Frame Operations - Analytics or Windowing Functions](http://wip.itversity.com/topic/data-frame-operations-analytics-or-windowing-functions-python/)
* [Spark SQL - Basic Transformations such as filtering, aggregations, joins etc](http://wip.itversity.com/topic/spark-sql-basic-transformations-such-as-filtering-aggregations-joins-etc/)
* [Spark SQL - Analytics Functions or Windowing Functions](http://wip.itversity.com/topic/spark-sql-analytics-functions-or-windowing-functions/)
* [Different file formats and dealing with custom delimiters](http://wip.itversity.com/topic/different-file-formats-and-dealing-with-custom-delimiters/)
* [Compression Concepts and Algorithms](http://wip.itversity.com/topic/compression-concepts-and-algorithms/)

# Quick revision of Python 3

posted on FEBRUARY 1, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

As part of this topic, let us quickly review the basic concepts of Python before jumping into Spark APIs. Python is a programming language and Spark APIs are compatible with Python (along with Scala, Java etc). It is imperative to master at least one of the programming languages to build applications using Spark.

Let us revise below concepts before jumping into pyspark (Spark with Python).

* Basics of Programming (help, type, indentation etc)
* Overview of Functions
* Lambda Functions
* Basic file I/O
* Collections and Map Reduce APIs
* Overview of Pandas Data Frames

We can use the jupyter notebook in the lab to revise python concepts.

## Basics of Programming

Let us talk about some of the basics of programming using Python 3.

* We can launch python CLI or use Jupyter Notebook to develop Python Code.
* **type** can be used to get the data type of the Python Variable or Object.
* **help** can be used as a CLI or as a function on Class or Object or a Function.
* We need to indent properly to define the scope while using Python for Programming.
* As Python is dynamically typed programming language we cannot specify data types while creating variable or objects. The type will be inherited based on the value assigned to a variable.
* It has all basic constructs such as if, while, for, the ternary operator etc.
* **Python** supports all basic data types as well as collections such as list, set, map etc.
* As part of the demo, we will see the usage of a type, help, basic program using the ternary operator as well as looping through a list (get even numbers from a list of elements)

## Overview of Functions

We need to revise the following related to functions.

* Pre-Defined Functions
  + Performing File I/O
  + String Manipulation Functions (will see few examples)
  + Date Manipulation Functions
  + Manipulating Collections
  + and more
* User Defined Functions
  + At times we need to develop new functions which are not available as part of Core Python or 3rd party Python modules.
  + Here are a few things we should recollect with respect to user-defined functions.
    - Function Specification (Function Name, Arguments, and Return type)
    - We can have a fixed number of arguments, varying number of arguments as well as keyword arguments for Functions in Python.
    - Function Definition or Logic
    - Return Statement
  + Functions can be passed as arguments to other functions.
  + We also will go through lambda functions in a separate topic.

## Lambda Functions

Let us revise the details related to Lambda Functions

* At times we might have to develop simple functions, especially to pass as an argument for higher order functions.
* In that case, we can use lambda functions.
* Lambda Functions are extensively used as part of modern programming languages.

|  |  |
| --- | --- |
|  | # Correct way of getting sumOfIntegers |
|  | def sumOfIntegers(lb, ub): |
|  | l = lb - 1 |
|  | return ((ub \* (ub + 1)) / 2) - ((l \* (l + 1)) / 2) |
|  |  |
|  | print(sumOfIntegers(2, 5)) |
|  |  |
|  | # To demonstrate lambda functions we will loop through the range |
|  | # Conventional approach, we need to write different functions for |
|  | # sum of range of numbers |
|  | # sum of squares in range of numbers |
|  | # and more |
|  | def sum(lb, ub): |
|  | total = 0 |
|  | for i in range(lb, ub + 1): |
|  | total += i |
|  | return total |
|  | print "sum of integers using conventional approach " + str(sum(3, 5)) |
|  |  |
|  | def sumOfSquares(lb, ub): |
|  | total = 0 |
|  | for i in range(lb, ub + 1): |
|  | total += (i \* i) |
|  | return total |
|  | print "sum of squares using conventional approach " + str(sumOfSquares(3, 5)) |
|  |  |
|  | # With lambda functions, we can get more concise and readable code |
|  | def sum(f, lb, ub): |
|  | total = 0 |
|  | for i in range(lb, ub + 1): |
|  | total += f(i) |
|  | return total |
|  | print "sum of integers using lambda functions " + str(sum(lambda i: i, 3, 5)) |
|  | print "sum of squares using lambda functions " + str(sum(lambda i: i \* i, 3, 5)) |
|  |  |
|  | # We can also pass named function as argument |
|  | def cube(i): return i \* i \* i |
|  | print "sum of cubes using lambda functions " + str(sum(lambda i: cube(i), 3, 5)) |

[**view raw**](https://gist.github.com/dgadiraju/7a91658c27b94305ffb42d01c5c868f2/raw/2741553cd75158f3eadbf05c1c5475fd2bf1c038/python-sum-lambda-functions.py)[**python-sum-lambda-functions.py**](https://gist.github.com/dgadiraju/7a91658c27b94305ffb42d01c5c868f2#file-python-sum-lambda-functions-py) hosted with  by [**GitHub**](https://github.com/)

## Basic File I/O

Let us see how we can read the data using Python File I/O APIs. We will limit the scope to read the data from a file into a collection.

* **open** is the API which facilitates us to create File Object
* We can perform **read()** to read the data from a file into the memory. When we apply read on files of text format, data will be loaded into memory as a string.
* We can load data at once or in iterations of multiple batches or buffers.
* To convert into the collection we can either use **split** or **splitlines**

## Collections and Map Reduce APIs

Now let us recollect details about collections and basic map reduce APIs.

* Python support 3 types of Collections
  + list – **[1, 2, 1, 5, 3]**
  + set – **{1, 2, 5, 3}**
  + dict – **{ ‘order\_id’: 1, ‘order\_date’: ‘2013-07-25 00:00:00.0’, ‘order\_customer\_id’: 1000, ‘order\_status’: ‘COMPLETE’ }**
  + a list is a heap of items while the set is a group of unique items
  + dict is similar to a hash map where keys are unique with corresponding value.
* We also have another data structure called Tuple. They are objects with unnamed attributes where values of attributes can be retrieved using positional notation
  + tuple – **(1, ‘2013-07-25 00:00:00.0’, 1000, ‘COMPLETE’)**
* Quite often we will create a list or set of tuples

#### Processing Collections

Let us see some simple examples

* Creating a list using orders data from a file
* Convert one element from the list into a tuple and perform tuple operations.
* Extract order\_dates from a list and get unique dates using set.
* Extract order\_id and order\_date as dict.

|  |  |
| --- | --- |
|  | orders = open('/data/retail\_db/orders/part-00000'). \ |
|  | read(). \ |
|  | splitlines() |
|  |  |
|  | # for order in orders[:10]: print(order) |
|  |  |
|  | orderDatesList = [] |
|  |  |
|  | for order in orders: |
|  | orderDatesList.append(order.split(',')[1]) |
|  |  |
|  | orderDates = set(orderDatesList) |
|  |  |
|  | # for order in list(orderDates)[:10]: print(order) |
|  |  |
|  | orderRecord = orders[0] |
|  | orderRecordElements = orders[0].split(',') |
|  |  |
|  | orderTuple = (int(orderRecordElements[0]), orderRecordElements[1], int(orderRecordElements[2]), orderRecordElements[3]) |
|  | # print(orderTuple[1]) |
|  |  |
|  | orderDict = {} |
|  |  |
|  | for order in orders: |
|  | orderDict[int(order.split(',')[0])] = order.split(',')[1] |
|  |  |
|  | print(orderDict[1]) |
|  | len(orderDict.keys()) |

[**view raw**](https://gist.github.com/dgadiraju/abb7bee2bcef0e3aa5042a75cdb13d3d/raw/805dc262320fcc89d5180dde25551ce95362ee47/BasicPythonCollectionOperations.py)[**BasicPythonCollectionOperations.py**](https://gist.github.com/dgadiraju/abb7bee2bcef0e3aa5042a75cdb13d3d#file-basicpythoncollectionoperations-py) hosted with  by [**GitHub**](https://github.com/)

#### Map Reduce APIs

Let us get into the details related to Map Reduce APIs to manipulate collections.

* We can process data in collections using different approaches – conventional loops, map-reduce etc.
* Map Reduce APIs such as filter, map etc take care of initializing the aggregator, looping through elements as well as returning the aggregator for us. We just need to focus on business logic.
* If we have to sort the collection then we need to convert the collection to list
* If we have to eliminate duplicates then we need to convert the collection to set
* Let us see how we can create a collection from a file and then apply map reduce APIs to compute revenue for a given order\_item\_order\_id.

|  |  |
| --- | --- |
|  | ordersPath = "C:\\Users\\dgadiraju\\Documents\\data-master\\retail\_db\\orders\\part-00000" |
|  | ordersFile = open(ordersPath) |
|  | ordersData = ordersFile.read() |
|  | orders = ordersData.splitlines() |
|  | for i in orders[:10]: |
|  | print(i) |

[**view raw**](https://gist.github.com/dgadiraju/0874f8bfff8b3642594137048fa897d1/raw/959e4630c7f8e9ec4d81c533275b65bf65d574f5/01-read-orders-data.py)[**01-read-orders-data.py**](https://gist.github.com/dgadiraju/0874f8bfff8b3642594137048fa897d1#file-01-read-orders-data-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ordersMap = map(lambda o: (o.split(",")[0], o.split(",")[3]), orders) |
|  | for i in list(ordersMap)[:10]: |
|  | print(i) |

[**view raw**](https://gist.github.com/dgadiraju/0874f8bfff8b3642594137048fa897d1/raw/959e4630c7f8e9ec4d81c533275b65bf65d574f5/02-get-orderid-and-status.py)[**02-get-orderid-and-status.py**](https://gist.github.com/dgadiraju/0874f8bfff8b3642594137048fa897d1#file-02-get-orderid-and-status-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | orderItemsPath = "C:\\Users\\dgadiraju\\Documents\\data-master\\retail\_db\\order\_items\\part-00000" |
|  | orderItemsFile = open(orderItemsPath) |
|  | orderItemsData = orderItemsFile.read() |
|  | orderItems = orderItemsData.splitlines() |
|  | for i in orderItems[:10]: |
|  | print(i) |

[**view raw**](https://gist.github.com/dgadiraju/0874f8bfff8b3642594137048fa897d1/raw/959e4630c7f8e9ec4d81c533275b65bf65d574f5/03-read-order-items.py)[**03-read-order-items.py**](https://gist.github.com/dgadiraju/0874f8bfff8b3642594137048fa897d1#file-03-read-order-items-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | orderItemsFiltered = filter(lambda oi: int(oi.split(",")[1]) == 2, orderItems) |
|  | orderItemsMap = map(lambda oi: float(oi.split(",")[4]), orderItemsFiltered) |
|  | #sum(orderItemsMap) |
|  | import functools as ft |
|  | ft.reduce(lambda x, y: x + y, orderItemsMap) |

[**view raw**](https://gist.github.com/dgadiraju/0874f8bfff8b3642594137048fa897d1/raw/959e4630c7f8e9ec4d81c533275b65bf65d574f5/04-get-revenue-for-given-order.py)[**04-get-revenue-for-given-order.py**](https://gist.github.com/dgadiraju/0874f8bfff8b3642594137048fa897d1#file-04-get-revenue-for-given-order-py) hosted with  by [**GitHub**](https://github.com/)

## Overview of Pandas Data Frames

While collections are typically the group of objects or tuples or simple strings, we need to parse them to further process the data. With Data Frames we can define the structure and we can reference values in each record using column names in Data Frames. Also, Data Frames provide rich and simple APIs to convert CSV Files into Data Frames and process them with developer-friendly API.

* Using read\_csv with names we can create Data Frame out of comma-separated data with the field name
* You can fetch data from specific columns using names
* We can filter data using query
* We can perform by key aggregations using group by and then aggregate functions
* We can also join data using align

Here are some of the examples of usage of Pandas data frames.

|  |  |
| --- | --- |
|  | orderItemsPath = "C:\\Users\\dgadiraju\\Documents\\data-master\\retail\_db\\order\_items\\part-00000" |
|  | orderItems = pd.read\_csv(orderItemsPath, names=["order\_item\_id", "order\_item\_order\_id", "order\_item\_product\_id", "order\_item\_quantity", "order\_item\_subtotal", "order\_item\_product\_price"]) |
|  | orderItems[['order\_item\_id', 'order\_item\_subtotal']] |
|  | orderItems.query('order\_item\_order\_id == 2') |
|  | orderItems.query('order\_item\_order\_id == 2')['order\_item\_subtotal'].sum() |
|  | orderItems.groupby(['order\_item\_order\_id'])['order\_item\_subtotal'].sum() |

[**view raw**](https://gist.github.com/dgadiraju/e7a16846ea96dc9dd8ab6aad6974565d/raw/885c50599b792b2dc191bf96e8e7e22bdffbfc2a/python-pandas-examples.py)[**python-pandas-examples.py**](https://gist.github.com/dgadiraju/e7a16846ea96dc9dd8ab6aad6974565d#file-python-pandas-examples-py) hosted with  by [**GitHub**](https://github.com/)

**Apache Spark – Getting Started**

posted on FEBRUARY 1, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

As part of this topic, we will review setup steps of Spark, understand different modules, Spark Architecture and how it is mapped to different execution modes such as YARN, Mesos etc.

Spark is nothing but distributed computing framework. To leverage the framework we need to learn APIs categorized into different modules and build applications using supported programming languages (like Scala, Python, Java etc).

* Setup Spark Environment
* Using ITVersity labs
* Spark Official Documentation
* Quick Review Of APIs
* Spark Modules
* Spark Data Structures
* Simple Application
* Spark Framework and Execution Modes
* DAG and Lazy Evaluation

**Setup Spark Environment**

Let us review the details related to setting up of Spark Environment. We have already covered how to setup environment earlier.

* Pre-requisites
  + 64 bit Computer
  + At least 4 GB RAM and enough storage
  + 64-bit Operating System – Windows 10, Linux, Mac etc
  + We would recommend Ubuntu on top of Window 10. You can either setup using Windows Subsystem for Linux or have virtual machine.
* Setup Process
  + Go to [https://spark.apache.org](https://spark.apache.org/)
  + Download the tarball of your choice
  + Uncompress and untar in your favorite location
  + Make sure to setup environment variables so that you can use commands such as spark-shell, pyspark, spark-submit from anywhere
* Understand the spark layout
* On top of Spark, we also need to have IDEs such as IntelliJ for Scala, Pycharm for Python to develop Spark based applications.
* Development Life Cycle
  + Develop using IDE
  + Build the code
* Once built, the code can be deployed in higher environments such as UAT, Production etc.

**Using ITVersity labs**

At ITVersity we provide multi-node Spark Cluster. Let us understand some important details with respect to ITVersity labs.

* Click [here](https://labs.itversity.com/) to visit our labs.
* We have 10+ node cluster in which Spark is integrated with Hadoop ecosystem
* It has a total capacity of 80+ cores and 400+ GB Memory and several terabytes of storage.
* The cluster is built using Hortonworks and cluster can be reviewed using Ambari.
* It has both Spark 1.6.3 and Spark 2.3.0
* You can launch Spark 2 either by exporting SPARK\_MAJOR\_VERSION to 2 or by using spark2 related commands.
  + spark2-shell (Spark 2 with Scala)
  + pyspark2 (Spark 2 with Python)
  + spark2-submit (Spark 2 submit command to submit Spark 2 jobs)

**Spark Official Documentation**

Let us understand how Spark Official Documentation is organized.

* Overview
* Quick Start
* Documentation for Different Modules
* API Docs
* Cluster Overview under Deploying
* Submitting Applications
* Execution Frameworks
* Tuning
* and more

**Quick Review Of APIs**

Let us have a quick review of Core APIs that are available in Spark. We will cover Data Frame APIs and Spark SQL at a later point in time.

* SparkContext exposes APIs such as textFile, sequenceFile to read data from files into a distributed collection called as RDD.
* RDD stands for Resilient Distributed Dataset and it is nothing but a distributed collection.
* It is typically loaded on to the executors created at the time of execution.
* RDD exposes APIs called as Transformations and Actions
* Transformations take one RDD as input and return another RDD as output while Actions trigger execution and get data into driver program.
* Examples of Transformations
  + Row Level Transformations – map, filter, flatMap etc
  + Aggregations – reduceByKey, aggregateByKey
  + Joins – join, leftOuterJoin, rightOuterJoin
  + Sorting – sortByKey
  + Ranking – groupByKey followed by flatMap with a lambda function
  + Except for Row Level Transformations, most of the other transformations have to go through the shuffle phase and trigger new stage.
  + Row Level Transformations are also known as Narrow Transformations.
  + Transformations that trigger shuffle and new stage are also called as Wide Transformations.
* Examples of Actions
  + Preview Data: take, takeSample, top, takeOrdered
  + Convert into Python List: collect
  + Total Aggregation: reduce, count
  + Writing into Files: saveAsTextFile, saveAsSequenceFile

**Spark Modules**

In the earlier versions of Spark, we have core API at the bottom and all the higher level modules work with core API. Examples of core API are a map, reduce, join, groupByKey etc. But with Spark 2, Data Frames and Spark SQL has become the core module.

* Core – Transformations and Actions – APIs such as map, reduce, join, filter etc. They typically work on RDD
* Spark SQL and Data Frames – APIs and Spark SQL interface for batch processing on top of Data Frames or Data Sets (not available for Python)
* Structured Streaming – APIs and Spark SQL interface for stream data processing on top of Data Frames
* Machine Learning Pipelines – Machine Learning data pipelines to apply Machine Learning algorithms on top of Data Frames
* GraphX Pipelines
* We can build applications using different programming languages such as Scala, Python, Java, R etc leveraging Spark APIs of the above-mentioned modules.

**Spark Data Structures**

We need to deal with 2 types of data structures in Spark – RDD and Data Frames.  We will see both in detail as we proceed further.

* RDD is there for quite some time and it is the low-level data structure which spark uses to distribute the data between tasks while data is being processed.
* RDD can be created using SparkContext APIs such as textFile.
* RDD will be divided into partitions while data being processed. Each partition will be processed by one task.
* The number of RDD partitions is typically based on HDFS block size which is 128 MB by default. We can control the number of minimum partitions by using additional arguments while invoking APIs such as textFile.
* Data Frame is nothing but RDD with the structure. We should be able to access the attributes of Data Frame using names.
* Typically we read data from file systems such as HDFS, S3, Azure Blob, Local file system etc
* Based on the file formats we need to use different APIs available in Spark to read data into RDD or Data Frame
* Spark uses HDFS APIs to read from, and/or write data to the underlying file system

**Simple Application**

Let us start with a simple application to understand details related to architecture using pyspark.

* As we have multiple versions of Spark on our lab and we are exploring Spark 2 we need to export SPARK\_MAJOR\_VERSION with 2
* For this demo, we will disable dynamic allocation by setting **spark.dynamicAllocation.enabled** to false.
* Launch pyspark using YARN and disabling dynamic allocation ( also, use spark.ui.port as well to specify unique port).
* Develop a simple word count program by reading data from /public/randomtextwriter/part-m-00000
* Save output to /user/training.

|  |  |
| --- | --- |
|  | data = sc.textFile('/public/randomtextwriter/part-m-00000') |
|  | wc = data. \ |
|  | flatMap(lambda line: line.split(' ')). \ |
|  | map(lambda word: (word, 1)). \ |
|  | reduceByKey(lambda x, y: x + y) |
|  | wc. \ |
|  | map(lambda rec: rec[0] + ',' + str(rec[1])). \ |
|  | saveAsTextFile('/user/training/core/wordcount') |

[**view raw**](https://gist.github.com/dgadiraju/02ded19303c6cb59dac0cd2199df85ec/raw/1f59692c0604db5d999a9d90b027719be7b1fcf9/pyspark-01-rdd-wordcount.py)[**pyspark-01-rdd-wordcount.py**](https://gist.github.com/dgadiraju/02ded19303c6cb59dac0cd2199df85ec#file-pyspark-01-rdd-wordcount-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | from pyspark.sql.functions import split, explode |
|  | data = spark.read.text('/public/randomtextwriter/part-m-00000') |
|  | wc = data.select(explode(split(data.value, ' ')).alias('words')). \ |
|  | groupBy('words'). \ |
|  | agg(count('words').alias('wc')) |
|  | wc.write.csv('/user/training/df/wordcount') |

[**view raw**](https://gist.github.com/dgadiraju/02ded19303c6cb59dac0cd2199df85ec/raw/1f59692c0604db5d999a9d90b027719be7b1fcf9/pyspark-02-df-wordcount.py)[**pyspark-02-df-wordcount.py**](https://gist.github.com/dgadiraju/02ded19303c6cb59dac0cd2199df85ec#file-pyspark-02-df-wordcount-py) hosted with  by [**GitHub**](https://github.com/)

Using this let us go through the Spark Framework.

**Spark Framework**

Let us understand the execution modes as well as different components of the Spark Framework. Also, we will recap some important aspects of YARN.

***Execution Modes***

Following are the different execution modes supported by Spark.

* Local (for development)
* Standalone (for development)
* Mesos
* YARN

As our cluster uses YARN, let us recap some important aspects of YARN.

* YARN uses Master (Resource Manager) and Slave (Node Managers) Architecture.
* YARN primarily takes care of resource management and scheduling the tasks.
* For each YARN Application, there will be an application master and set of containers created to process the data.
* We can plugin different distributed frameworks into YARN, such as Map Reduce, Spark etc.
* Spark creates executors to process the data and these executors will be managed by Resource Manager and per job Application Master.

***Execution Framework***

Let us understand the Spark Execution Framework by running wordcount program using RDD.

* Driver Program
* Spark Context
* Executors
* Executor Cache
* Executor Tasks
* Job
* Stage
* Task (Executor Tasks)

**Directed Acyclic Graph and Lazy Evaluation**

There are many APIs in Spark. But most of the APIs do not trigger the execution of Spark job.

* When we create a Spark Context object it will procure resources from the cluster.
* APIs used to read the data such as textFile as well as to process the data such as map, reduce, filter etc does not trigger immediate execution. They create variables of type RDD which also point to DAG.
* They run in driver program and build DAG. DAG will tell how it should execute. Each variable has a DAG associated with it.
* When APIs which are categorized as action (such as take, collect, saveAsTextFile) are used DAG associated with the variable is executed.
* We can look at the DAG details by using toDebugString on top of the variables created.
* We can visualize DAG as part of Spark UI.

# RDD, Data Frame, DAG and Lazy Evaluation

posted on FEBRUARY 1, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

Now let us see details about data structures in Spark such as Resilient Distributed Datasets, Data Frames, Directed Acyclic Graph, Lazy Evaluation etc.

* Difference between Python list and RDD
* Resilient Distributed Datasets
* Data Frames
* Overview of Transformations and Actions
* Directed Acyclic Graph and Lazy Evaluation

## Difference between Python list and RDD

Let us create simple RDD and compare with Python list.

* Python list is nothing but a heap of elements.
* We can manipulate Python list using APIs such as map, filter, reduce, set, sort, sorted etc.
* Python list is typically processed in a linear fashion
* RDD stands for Resilient Distributed Dataset. It is nothing but a distributed list.
* RDD is a distributed collection or list provided as part of Spark.
* APIs on RDD facilitate us to process it in a distributed fashion.
* We can manipulate Spark RDD using RDD functions such as map, flatMap, filter, reduceByKey etc.
* Spark gives us the desired scalability as APIs will process RDD in distributed fashion.

## Resilient Distributed Datasets

Resilient Distributed Datasets (in short RDD) is the fundamental data structure in Spark.

#### Creation of RDD

Let us see how we can create RDD.

* On top of SparkContext (sc) we have several APIs to create RDD using data from files.
  + textFile
  + sequenceFile
  + Hadoop related APIs
* These APIs will not trigger execution immediately. They will update the DAG and we need to perform actions to trigger execution.
* We can use APIs such as **take(10)** to preview the data from RDD. We cannot perform traditional list operations such **[:10]** on RDD.
* Characteristics of RDD
  + In-memory
  + Distributed
  + Resilient

#### Execution Life Cycle

* Data from files will be divided into RDD partitions and each partition is processed by a separate task
* By default, it will use HDFS block size (128 MB) to determine the partition si
* We can increase (cannot decrease) number of partitions by using an additional parameter in sc.textFile
* By default when data is loaded into memory each record will be serialized into Java object

#### RDD Persistence

Typically data will not be loaded into memory immediately when we create RDD as part of the program. It will be processed in real time by loading data into memory as it is processed. If we have to retain RDD in memory for an extended period of time, then we have to use RDD Persistence.

* Let us see what happens when RDD is loaded into memory
  + Serialize into Java Objects
  + Get into memory
  + As data is processed RDD  partitions will be flushed out of memory as tasks are completed.
* We can persist the RDD partitions at different storage levels
  + MEMORY\_ONLY (default)
  + MEMORY\_AND\_DISK
  + DISK\_ONLY
  + and more

## Data Frames

Many times data will have structure. Using RDD and then core APIs is some what tedious and cryptic. We can use Data Frames to address these issues. Here are the some of the advantages using Data Frames

* Flexible APIs (Data Frame native operations as well as SQL)
* Code will be readable
* Better organized and manageable
* Uses latest optimizers
* Process data in binary format
* Can generate execution plans based on statistics collected (for permanent tables such as Hive tables)

Data Frames are nothing but RDD with structure. Once Data Frame is created we can refer attributes or columns using names.

We will talk about processing data using Data Frames in the next chapter. For now, we will be focusing on Core APIs

## Overview of Transformations and Actions

Spark Core APIs are categorized into Transformations and Actions. Let us explore them at a higher level. These can be performed using APIs or Spark SQL on top of Data Frames.

* Transformations
  + Row-level transformations – map, flatMap, filter
  + Joins – join, leftOuterJoin, rightOuterJoin
  + Aggregations – reduceByKey, aggregateByKey
  + Sorting data – sortByKey
  + Group operations such as ranking – groupByKey
  + Set operations – union, intersection
  + and more
* Actions
  + Previewing data – first, take, takeSample
  + Converting RDD into the typical collection – collect
  + Total aggregations – count, reduce
  + Total ranking – top
  + Saving files – saveAsTextFile, saveAsNewAPIHadoopFile etc
  + and more

Transformations are the APIs which take RDD as input and return another RDD as output. These APIs does not trigger execution but update the DAG. Actions take RDD as input and return a primitive data type or regular collection to the driver program.

Also, we can use actions to save the output to the files. Actions trigger execution of DAG.

## Directed Acyclic Graph and Lazy Evaluation

There are many APIs in Spark. But most of the APIs do not trigger the execution of Spark job.

* When we create a Spark Context object it will procure resources from the cluster
* APIs used to read the data such as textFile as well as to process the data such as map, reduce, filter etc does not trigger immediate execution. They create variables of type RDD which also point to DAG.
* They run in driver program and build DAG. DAG will tell how it should execute. Each variable has a DAG associated with it.
* When APIs which are categorized as action (such as take, collect, saveAsTextFile) are used DAG associated with the variable is executed.
* In Scala, we can look at the DAG details by using toDebugString on top of the variables created.
* We can visualize DAG as part of Spark UI

[← Previous Topic](http://wip.itversity.com/topic/apache-spark-getting-started/)

# Transformations and Actions

posted on FEBRUARY 1, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

Let us look into how we can perform basic transformations such as row-level transformations, aggregations, joins, sorting etc as part of this topic.

* Data Processing Life Cycle
* Sum Of Even Numbers
* Develop Word Count Program
* Get Daily Revenue
* Reading the Data
* Apply Filtering
* Joining Data Sets
* Aggregations
* Sorting Data
* Saving and Conclusion

## Data Processing Life Cycle

Let us understand the typical Data Processing Life Cycle and corresponding Spark APIs that can be used in different phases of the life cycle.

Data Processing Life Cycle can be divided into three main phases

* Reading the data
* Processing the data
* Writing the data

#### Reading the data

Read data from the file system using appropriate APIs using a SparkContext object.

* **textFile** to read text data.
* **sequenceFile** to read data in sequence file format.
* **newAPIHadoopRDD**
* **newAPIHadoopFile**
* **hadoopFile**
* **hadoopRDD**
* As part of Hadoop APIs, we got classes for different input file formats such as text, sequence, key-value, nline etc to read the data.
* **textFile** and **sequenceFile** are nothing but wrappers on top of existing HDFS Classes to read text and sequence files. To read other file formats, we need to use other HDFS based APIs highlighted above.
* **newAPIHadoopRDD** and **newAPIHadoopFile** are built on top of new Hadoop APIs while **hadoopFile** and **hadoopRDD** are built on old Hadoop APIs.

#### Processing the Data

We can process data by applying different transformations for different purposes.

* Row-level transformations – map, flatMap
* Filtering – filter
* Aggregations – reduceByKey and aggregateByKey
* Joins – performing inner joins and outer joins
* Sorting data – sortByKey
* Ranking – groupByKey followed by flatMap

#### Writing the Data

We can Write or Save data to File System by using appropriate write APIs on top of RDDs.

* **saveAsTextFile** to read text data.
* **saveAsSequenceFile** to read data in sequence file format.
* **saveAsNewAPIHadoopRDD**
* **saveAsNewAPIHadoopFile**
* **saveAsHadoopFile**
* **saveAsHadoopRDD**
* As part of HDFS APIs, we got classes for different output file formats such as text, sequence, key-value etc to write the data.
* **saveAsTextFile** and **saveAsSequenceFile** are nothing but wrappers on top of existing HDFS Classes to write data into text and sequence files. To write into file formats, we need to use other Hadoop APIs highlighted above.
* **saveAsNewAPIHadoopRDD** and **saveAsNewwAPIHadoopFile** are built on top of new Hadoop APIs while **saveAsHadoopFile** and **saveAsHadoopRDD** are built on old Hadoop APIs.
* Process Data by applying the required transformations.
* Row-level transformations and filter are also known as Narrow Transformations.
* Joins, Aggregations, Sorting and other transformations are also known as Wide Transformations.
* Wide Transformations typically result in a new stage and data will be shuffled.

## Sum Of Even Numbers

Let us understand how we can convert a collection to RDD and perform RDD operations to process. As part of SparkContext, we have an API called parallelize, which can be used to convert the typical list into RDD. Similarly, we can use collect API on RDD to convert into the list.

* Create a list from 1 to 100000 using range – l = list(range(1, 100001))
* Convert into RDD – lRDD = sc.parallelize(l)
* Filter for even numbers – lEven = lRDD.filter(lambda n: n % 2 == 0)
* Get sum of the even numbers – sumEven = lEven.reduce(lambda x, y: x + y)
* collect() will get the data from RDD to driver program as a list. Do not use collect() to preview the data, it can run into out of memory issues.

## Develop Word Count Program

Let us develop a word count program using pyspark. As part of this example, we will see flatMap, map, and reduceByKey.

* flatMap – convert a single record into multiple records based upon the logic. The number of records in output RDD will be greater than the number of records in input RDD.
* map – apply the transformation on individual records which will result in a changed value. The number of records in both input RDD and output RDD will be the same.
* reduceByKey – apply the transformation in the key, where all the values related to a particular key is grouped and then the logic passed as lambda function is applied. This results in shuffling and hence you will see a new stage.
* We will also use actions such as take, count etc to preview data as well as to validate results.
* Problem Statement – For unique word in input file we need to get how many times it is repeated. The input file contains a bunch of lines with words.
* Design
  + Break each line into words (using flatMap). If you want to convert each record into multiple records based on logic we need to use flatMap API. flatMap take lambda function as an argument for which we need to pass logic to break down input record into an array and flatMap inbuilt logic will return each element in the array as a record.
  + As we broke each line into word, we need to convert them into tuples (using map). It will facilitate us to use by key operations such as reduceByKey.
  + Paired RDD (output of map function) can now be passed to reduceByKey and get the count for each word.
  + Logic passed to as part of reduceByKey execute both on the map output as well as reduce input.

|  |  |
| --- | --- |
|  | lines = sc.textFile("/public/randomtextwriter/part-m-00000") |
|  | words = lines.flatMap(lambda line: line.split(" ")) |
|  | wordTuples = words.map(lambda word: (word, 1)) |
|  | wordCount = wordTuples.reduceByKey(lambda x, y: x + y) |
|  | wordCount.saveAsTextFile("/user/training/bootcamp/pyspark/wordcount") |

[**view raw**](https://gist.github.com/dgadiraju/0cb7b3b80100659c3e66cbd44f9e97fc/raw/b834da71201d61e0ab904ae65d0d5437b0c1be92/pyspark-wordcount.py)[**pyspark-wordcount.py**](https://gist.github.com/dgadiraju/0cb7b3b80100659c3e66cbd44f9e97fc#file-pyspark-wordcount-py) hosted with  by [**GitHub**](https://github.com/)

## Get Daily Revenue

Let us develop solution to get revenue for each day considering completed orders. As part of this example, we will explore the distinct, filter, map, join, reduceByKey, sortByKey, saveAsTextFile etc. We will explore APIs in depth while coming up solution for the problem statement.

First, let us start with problem statement and design before getting into coding.

#### Problem Statement

Let us first define problem statement.

* Our hypothetical client is an eCommerce retail platform for apparel.
* Someone can go to that platform and start placing the orders.
* Executive management wants to see revenue-related reports based on the date such as daily, weekly, monthly etc.
* We only need to consider complete or closed orders to compute revenue.

#### High-level Design

Let us map the requirements to the data model and come up with the design before we get into coding.

* Our retail data set have 6 tables – orders, order\_items, products, categories, departments, and customers.
* We can review the relationships by going through the data model published [here](https://www.cloudera.com/developers/get-started-with-hadoop-tutorial/exercise-1.html).
* orders have order level attributes such as **order\_id**, order\_date, order\_customer\_id and order\_status.
* order\_items have item level details such as order\_item\_id, **order\_item\_order\_id**, order\_item\_product\_id, order\_item\_quantity, order\_item\_subtotal and order\_item\_product\_price.
* As the lowest granularity for our report is daily, we will compute daily revenue. Other reports can be derived from it using any standard BI tool.

#### Design Outcome

Outcome of technical design will look like this.

* Read the data – orders, and order\_items
* Filter for completed and closed orders from orders using order\_status
* Join filtered orders with order\_items and get order\_date from orders and order\_item\_subtotal from order\_items
* Aggregate and compute daily revenue using order\_date as key
* Sort the data based on the date.
* Save the output to file system in the form of text file. Data should be delimited using a comma **(,)**.

## Reading the Data

As we have briefly gone through the APIs to read the data using sc, let us understand the details about data set and use appropriate APIs to read both orders and order\_items.

* Let us read orders data orders = sc.textFile("/public/retail\_db/orders")
* Let us read order items data  orderItems = sc.textFile("/public/retail\_db/order\_items")
* As part of data analysis, we will see what all different statuses we have in orders. For that, we need to read orders data into RDD, extract order\_status and apply distinct on top of it orders.map(lambda order: order.split(",")[3]).distinct().collect()

## Apply Filtering

As we understood the data, we will see how we can filter for completed or closed orders using filter transformation.

* It creates new RDD for the records which satisfy the criteria passed as an argument. We need to pass the filter criteria as a function to filter.
* As the filter is transformation, it will not trigger execution immediately.
* Let us see few examples.
  + Get orders which are in COMPLETE status
  + Get orders which are in COMPLETE or CLOSED status
  + Get orders whose status have PENDING
  + Get orders which are placed in the month of 2014 January
  + Get orders which are placed in the month of 2014 January and in COMPLETE or CLOSED status
* Here is how we can get orders which are either in COMPLETE or CLOSED status – ordersFiltered = orders.filter(lambda o: o.split(",")[3] in ("COMPLETE", "CLOSED"))

## Joining Data Sets

Let us understand how to perform joins between 2 data sets. Once we learn the concepts we will join ordersFiltered with orderItems.

* join – It can be used to join multiple data sets on a common key. It can be performed on the RDDs where each element in the form of (k, v) and (k, w). It results in new RDD where each element is in the form of (k, (v, w)).
* Typically we use the map to transform both our input data sets into key-value pairs.
  + orders data – ordersMap = orders.map(lambda o: (int(o.split(",")[0]), o)
  + order items data – orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), oi))
  + Once data sets are in the required format, joining is very straight forward
  + Key in both the data sets should have the same business context (typically primary key in one table and foreign key in another table)
  + joining data – ordersJoin = ordersMap.join(orderItemsMap)
* outer join – APIs also support outer joins (leftOuterJoin, rightOuterJoin, and fullOuterJoin). left and right are functionally and purely based on the position of the parent data set (in our case it is **orders** which drives the outer join)
* Get orders with no corresponding order items – ordersMap.leftOuterJoin(orderItemsMap).filter(lambda o: o[1][1] == None)

#### Get order\_date and order\_item\_subtotal

As we understood how to perform joins let us go ahead and see how we can get order\_date and order\_item\_subtotal from our input data sets using join.

* Read both orders and order\_items (orderItems)
* Apply filter on orders (ordersFiltered)
* Apply map on both ordersFiltered and orderItems – ordersFilteredMap and orderItemsMap
  + orders data – ordersFilteredMap = ordersFiltered.map(lambda o: (int(o.split(",")[0]), o.split(",")[1]))
  + order items data – orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.split(",")[4])))
* Perform join on the paired RDDs – ordersJoin = ordersFilteredMap.join(orderItemsMap)
* ordersJoin have the data which contain (order\_id, (order\_date, order\_item\_subtotal))
* We don’t need order\_id any more and hence we can apply map and eliminate order\_id
* Discarding order id from join results – ordersJoinMap = ordersJoin.map(lambda o: o[1])

## Aggregations

Once we join ordersFiltered and orderItems, let us see how we can aggregate the data. But before finalizing the logic, let us explore different options that are available.

* We typically use reduceByKey for aggregations.
* Even though we can achieve this using aggregateByKey and groupByKey, reduceByKey is a more appropriate way.
  + For aggregations, we should not prefer using groupByKey as we cannot pass the function which takes care of the aggregation.
  + groupByKey only takes care of grouping the data based on the key. For aggregation, we have to use a map on the output of groupByKey.
  + Both reduceByKey and aggregateByKey takes care of aggregation on top of grouping the data per key.
  + Let us first perform the tasks using reduceByKey. We will see examples for aggregateByKey at a later point in time.
* Tasks using reduceByKey
  + Get number of orders placed per day
  + Get number of orders by status placed per day
  + Get order\_revenue for each order using order\_items
  + Get order\_revenue as well as item\_count for each order using order\_items
* Criteria for reduceByKey
  + Used to perform aggregations
  + The value of each of the element in RDD can be either numeric or a tuple with numeric elements.
  + The elements in input RDD should be of type tuple and it should be a pair.
  + The datatype of each element in input RDD and output RDD should be the same.
  + Compute average revenue per order item for each order. We can compute average revenue using this formula – sum(order\_item\_subtotal) per order/sum(order\_item\_quantity) per order.
* We also have reduceByKeyLocally, however, final aggregation will be done in the driver program. You can use there are not many unique keys on which aggregation is being performed.

#### Get Daily Revenue

As we have seen several examples with respect to reduceByKey, let us use it to get daily revenue as part of the solution for our problem statement.

* Read both orders and order\_items (orderItems)
* Apply filter on orders (ordersFiltered)
* Apply map on both ordersFiltered and orderItems – ordersFilteredMap and orderItemsMap
  + orders data – ordersFilteredMap = ordersFiltered.map(lambda o: (int(o.split(",")[0]), o.split(",")[1]))
  + order items data – orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.split(",")[4])))
* Perform join on the paired RDDs – ordersJoin = ordersFilteredMap.join(orderItemsMap)
* ordersJoin have the data which contain (order\_id, (order\_date, order\_item\_subtotal))
* We don’t need order\_id any more and hence we can apply map and eliminate order\_id
* Discarding order id from join results – ordersJoinMap = ordersJoin.map(lambda o: o[1])
* As ordersJoinMap have a date as key and item subtotal as value, we can use reduceByKey to get daily revenue.
* Get daily revenue – dailyRevenue = ordersJoinMap.reduceByKey(lambda x, y: x + y)

## Sorting Data

As we got aggregated results we might want to sort before saving into file. Let us look into how we can perform sorting leveraging APIs on top of RDD.

* There are 2 functions to sort the data – sortBy and sortByKey. Both are transformations. They take RDD as input and return sorted RDD based on criteria as output.
* sortBy is to sort RDDs which do not have elements in the form of tuples with 2 elements while sortByKey is to sort paired RDDs based on the first element.
* We can pass RDD which contain tuples of 2 elements to sortByKey and sort the data.
* Key in each of the tuple can be of a primitive type or tuple.
* By default data is sorted in ascending order as ascending is by default true. We can pass ascending as false to sort the data in descending order.
* We can also pass a custom key to sortByKey as a function to keyword argument keyfunc.
* The output of sortByKey is again an RDD which contain tuples.
* If we want to save the data after sorting, typically we use the map function to discard the key and apply the necessary transformation to use delimiters before saving into a file.

#### Tasks

Let us see few tasks about sorting the data. We will perform all 3 tasks using sortBy as well as sortByKey.

* Sort orders data by order\_date. The value should be the whole information about each order.
* Sort orders data by order\_date and then order\_customer\_id. The value should be the whole information about each order.
* Sort order\_items data by order\_item\_order\_id and then order\_item\_subtotal descending. The value should contain the whole order\_item.
* We will also sort dailyRevenue in ascending order by date – dailyRevenueSorted = dailyRevenue.sortByKey()

#### Get Daily Revenue Sorted

As we have understood how we can sort the data using sortBy as well as sortByKey, let us see how we can sort the daily revenue by date as part of the solution for our problem statement.

* Read both orders and order\_items (orderItems)
* Apply filter on orders (ordersFiltered)
* Apply map on both ordersFiltered and orderItems – ordersFilteredMap and orderItemsMap
  + orders data – ordersFilteredMap = ordersFiltered.map(lambda o: (int(o.split(",")[0]), o.split(",")[1]))
  + order items data – orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.split(",")[4])))
* Perform join on the paired RDDs – ordersJoin = ordersFilteredMap.join(orderItemsMap)
* ordersJoin have the data which contain (order\_id, (order\_date, order\_item\_subtotal))
* We don’t need order\_id any more and hence we can apply map and eliminate order\_id
* Discarding order id from join results – ordersJoinMap = ordersJoin.map(lambda o: o[1])
* As ordersJoinMap have a date as key and item subtotal as value, we can use reduceByKey to get daily revenue.
* Get daily revenue – dailyRevenue = ordersJoinMap.reduceByKey(lambda x, y: x + y)
* We can sort dailyRevenue in ascending order by date using sortByKey as our input RDD is paired RDD which contain date as a key. – dailyRevenueSorted = dailyRevenue.sortByKey()
* If we want to sort the data in descending order by revenue, we can use sortBy – dailyRevenue.sortBy(lambda rec: -rec[1])

## Saving and Conclusion

As we are done with processing the data, now it is time for us to save the data.

* We can perform an action such as saveAsTextFile to save the output. Typically we transform our data to the way it is supposed to be saved (e. g: Delimiters) – dailyRevenueSortedMap = dailyRevenueSorted.map(lambda oi: oi[0] + "," + str(oi[1]))
* Saving output – dailyRevenueSorted.saveAsTextFile("/user/training/bootcamp/pyspark/daily\_revenue")
* We can also save the data in Sequence File or other Hadoop supported File formats using appropriate APIs. Using Data Frames, we can easily save the data in industry file formats such as orc, parquet, avro etc. We will see those examples while exploring Data Frames in detail.

|  |  |
| --- | --- |
|  | orders = sc.textFile("/public/retail\_db/orders") |
|  | orderItems = sc.textFile("/public/retail\_db/order\_items") |
|  | ordersFiltered = orders.filter(lambda o: o.split(",")[3] in ("COMPLETE", "CLOSED")) |
|  | ordersFilteredMap = ordersFiltered.map(lambda o: (int(o.split(",")[0]), o.split(",")[1])) |
|  | orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.split(",")[4]))) |
|  | ordersJoin = ordersFilteredMap.join(orderItemsMap) |
|  | ordersJoinMap = ordersJoin.map(lambda o: o[1]) |
|  | dailyRevenue = ordersJoinMap.reduceByKey(lambda x, y: x + y) |
|  | dailyRevenueSorted = dailyRevenue.sortByKey() |
|  | dailyRevenueSortedMap = dailyRevenueSorted.map(lambda oi: oi[0] + "," + str(oi[1])) |
|  | dailyRevenueSortedMap.saveAsTextFile("/user/training/bootcamp/pyspark/daily\_revenue") |

[**view raw**](https://gist.github.com/dgadiraju/7e3649bc7eb84e48460495b0c1007a25/raw/6f388db551387e1e031dbd2a8caeed8fd4c65e11/pyspark-get-daily-revenue.py)[**pyspark-get-daily-revenue.py**](https://gist.github.com/dgadiraju/7e3649bc7eb84e48460495b0c1007a25#file-pyspark-get-daily-revenue-py) hosted with  by [**GitHub**](https://github.com/)

# Transformations and Actions – Continued

posted on FEBRUARY 21, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

As we understand basic transformations such as map, flatMap, reduce etc, now let us look at few advanced operations.

* Overview of Shuffling
* Using aggregateByKey
* mapPartitions
* Global Ranking
* Ranking using groupByKey

## Overview of Shuffling

Let us understand the concept of Shuffling.

* As we have seen a Spark job will run in multiple stages
* Stages will run in a linear fashion. For example, Stage 1 will run only after Stage 0 is completely done
* In each stage, data will be processed using tasks
* The output of stage 0 tasks will be passed as input to stage 1 tasks
* When the output of tasks in earlier stages is passed as input to tasks in later stages, the following happens
  + Data will be partitioned by using a hash mod algorithm
  + Data related to keys will be grouped together
  + This data will be cached in memory and it might be spilled to disk as well.
  + Data related to a particular key from all tasks of earlier stages will be passed to one task in later stages.
  + This entire process is called shuffling
  + When certain APIs such as reduceByKey/aggregateByKey is used, it will also perform something called combining which can improve the performance significantly.
  + APIs such as join, reduceByKey, aggregateByKey, groupByKey etc result in shuffling.
* The number of tasks in subsequent stages is determined by one of these
  + Number of partitions from an earlier stage
  + numTasks or numPartitions argument as part of APIs that result in shuffling
  + repartition or coalesce (covered as part of next topic)
* An accurate number of tasks can only be determined after understanding data behavior in detail. Here are some of the criteria
  + The ratio between input data vs. output (in case of filtering and aggregations output size will be considerably lower)
  + Keys on which data is shuffled (sparse keys vs. dense keys)
  + Joins and potential cartesian products
  + and more

Here are the examples of groupByKey, reduceByKey and aggregateByKey to understand the differences.

|  |  |
| --- | --- |
|  | orderItems = sc.textFile("/public/retail\_db/order\_items") |
|  | orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.split(",")[4]))) |
|  | orderItemsGBK = orderItemsMap.groupByKey(3) |
|  | orderItemsGBKMap = orderItemsGBK.map(lambda oi: (oi[0], sum(oi[1]))) |
|  | for i in orderItemsGBKMap.take(10): print(i) |

[**view raw**](https://gist.github.com/dgadiraju/9fd369bb050034f229f3a9e1153059a9/raw/fbed222cf24c7fd5868bb794a3c0c0e7cca24a4a/01-pyspark-aggregations-groupbykey.py)[**01-pyspark-aggregations-groupbykey.py**](https://gist.github.com/dgadiraju/9fd369bb050034f229f3a9e1153059a9#file-01-pyspark-aggregations-groupbykey-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | orderItems = sc.textFile("/public/retail\_db/order\_items") |
|  | orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.split(",")[4]))) |
|  | orderItemsRBK = orderItemsMap.reduceByKey(lambda x, y: x + y, 3) |
|  | for i in orderItemsRBK.take(10): print(i) |

[**view raw**](https://gist.github.com/dgadiraju/9fd369bb050034f229f3a9e1153059a9/raw/fbed222cf24c7fd5868bb794a3c0c0e7cca24a4a/02-pyspark-aggregations-reducebykey.py)[**02-pyspark-aggregations-reducebykey.py**](https://gist.github.com/dgadiraju/9fd369bb050034f229f3a9e1153059a9#file-02-pyspark-aggregations-reducebykey-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | orderItems = sc.textFile("/public/retail\_db/order\_items") |
|  | orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.split(",")[4]))) |
|  | orderItemsABK = orderItemsMap.aggregateByKey((0.0, 0), |
|  | lambda x, y: (x[0] + y, x[1] + 1), |
|  | lambda x, y: (x[0] + y[0], x[1] + y[1]), |
|  | 3 |
|  | ) |
|  | for i in orderItemsABK.take(10): print(i) |

[**view raw**](https://gist.github.com/dgadiraju/9fd369bb050034f229f3a9e1153059a9/raw/fbed222cf24c7fd5868bb794a3c0c0e7cca24a4a/03-pyspark-aggregations-aggregatebykey.py)[**03-pyspark-aggregations-aggregatebykey.py**](https://gist.github.com/dgadiraju/9fd369bb050034f229f3a9e1153059a9#file-03-pyspark-aggregations-aggregatebykey-py) hosted with  by [**GitHub**](https://github.com/)

## Using aggregateByKey

Let us understand what is aggregateByKey how we can leverage it for some of the aggregation problems.

* Data Processing Cycle – Narrow Transformations -> Shuffling -> Narrow Transformations
* As part of shuffling, typically data is partitioned and grouped based on the key.
* If we use reduceByKey or aggregateByKey, on top of partitioning and grouping it also takes care of aggregations.
* Aggregations are typically done in both the stages (seqOp/seqFunc and combOp/combFunc) as part of shuffling.
* If the logic for both the stages is same, then we use reduceByKey otherwise we use aggregateByKey.
* Using aggregateByKey
  + It takes 3 arguments – zeroValue, seqFunc and combFunc
  + zeroValue is primarily used for initializing the aggregator.
  + seqFunc should have the logic to compute intermediate aggregated values.
  + combFunc should have the logic to compute final values
  + The data type of elements in input RDD and output RDD need not to be the same.
  + seqFunc and combFunc logic can be slightly different.
* Tasks
  + Get order revenue as well as count from order\_items
  + Compute order revenue using order\_item\_product\_price and order\_item\_quantity

## mapPartitions

APIs such as map, filter, flatMap work on individual records. We can implement any of this functionality using mapPartitions, but the difference is in its execution.

* For map, filter, flatMap – the number of executions of a lambda function is equal to the number of records
* For mapPartitions – the number of executions of a lambda function is equal to the number of partitions
* As part of the lambda function in mapPartitions
  + Process data as a collection
  + Apply Python map or filter or flatten
  + Return a collection
* The elements from the collection returned from lambda function will be added to RDD
* Use cases where mapPartitions can perform better – Looking up into a database. Instead of creating a connection for each record, we can establish a connection once per for each partition (if looking up into the database is required as part of data processing)
* Here is the example of getting word count using mapPartitions

|  |  |
| --- | --- |
|  | lines = sc.textFile("/public/randomtextwriter/part-m-00000") |
|  | def getWordTuples(i): |
|  | import itertools as it |
|  | wordTuples = map(lambda s: (s, 1), it.chain.from\_iterable(map(lambda s: s.split(" "), i))) |
|  | return wordTuples |
|  |  |
|  | wordTuples = lines.mapPartitions(lambda i: getWordTuples(i)) |
|  | for i in wordTuples.reduceByKey(lambda x, y: x + y).take(10): |
|  | print(i) |

[**view raw**](https://gist.github.com/dgadiraju/1dd959c02ea0e09b5cbfa62287c4cb31/raw/89b9a7b079fa3f5d69c68f828e0cd59ce011efe2/pyspark-wordcount-mapPartitions.py)[**pyspark-wordcount-mapPartitions.py**](https://gist.github.com/dgadiraju/1dd959c02ea0e09b5cbfa62287c4cb31#file-pyspark-wordcount-mappartitions-py) hosted with  by [**GitHub**](https://github.com/)

## Global Ranking

Spark provides a few actions for global ranking.

* top (equivalent to sortByKey and then take)
* takeOrdered
* As these are actions, they will trigger execution and results will be returned to the driver program as a collection.
* We will not be able to save the data using APIs such as saveAsTextFile as the output is a typical Python list rather than RDD.

## Ranking using groupByKey

groupByKey is a very powerful API which groups the values based on the key. It can be used to solve problems such as ranking.

* Task 1: Get top N products by price in each category
  + Let us read products data into RDD
  + Convert the data to (k, v) using product category id as key and the entire product record as a value
  + Use groupByKey
  + Use first and get the first record and read the second element to regular python collection variable (productsPerCategory)
  + Develop a function to get top N products by price in that list
  + Validate the function using productsPerCategory
  + Invoke the function on the output of groupByKey as part of flatMap

|  |  |
| --- | --- |
|  | products = sc.textFile("/public/retail\_db/products") |
|  | productsFiltered = products.filter(lambda p: p.split(",")[4] != "") |
|  | productsMap = productsFiltered.map(lambda p: (int(p.split(",")[1]), p)) |
|  | productsGBCategory = productsMap.groupByKey() |
|  |  |
|  | # p = list(productsGBCategory.first()[1]) |
|  |  |
|  | def getTopNProducts(products, topN): |
|  | return sorted(products, key=lambda k: float(k.split(",")[4]), reverse=True)[:topN] |
|  |  |
|  | # getTopNProducts(p, 3) |
|  |  |
|  | topNProductsByCategory = productsGBCategory.flatMap(lambda p: getTopNProducts(list(p[1]), 3)) |
|  | for i in topNProductsByCategory.take(10): |
|  | print(i) |

[**view raw**](https://gist.github.com/dgadiraju/7d2a6e3ad9e7f656e5bbe868946b7d95/raw/3f079b6fde1d298ea15ba32c19088a148dbddba2/groupByKey-getTopNProductsPerCategory.py)[**groupByKey-getTopNProductsPerCategory.py**](https://gist.github.com/dgadiraju/7d2a6e3ad9e7f656e5bbe868946b7d95#file-groupbykey-gettopnproductspercategory-py) hosted with  by [**GitHub**](https://github.com/)

* Task 2: Get top N Priced products in each category
  + Let us read products data into RDD
  + Convert the data to (k, v) using product category id as key and the entire product record as a value
  + Use groupByKey
  + Use first and get the first record and read the second element to regular python collection variable (productsPerCategory)

## repartition and coalesce

Now let us understand how we can control number of tasks to process data after first stage.

* Each of the APIs which result in shuffling have additional argument numKeys or numPartitions
* We can use repartition to control the number of tasks or partitions in subsequent stages
* repartition result in shuffling process
* We can increase or decrease number of partitions in RDD using repartition
* coalesce can only be used to reduce number of partitions
* coalesce does not result in shuffling process
* Here is the example which covers coalesce and repartition (repartition will be slow in this case as it generate additional stage for shuffling data into new partitions)

|  |  |
| --- | --- |
|  | lines = sc.textFile("/public/randomtextwriter/part-m-00000", 18) |
|  | words = lines.flatMap(lambda line: line.split(" ")) |
|  | wordTuples = words.map(lambda word: (word, 1)) |
|  | wc = wordTuples.reduceByKey(lambda x, y: x + y, 36) |
|  | wc.coalesce(4).saveAsTextFile("/user/training/bootcamp/pyspark/wordcount02") |

[**view raw**](https://gist.github.com/dgadiraju/d6a920b806007dc49450293fae45beef/raw/7914a6a57e64092b0df2b3efbdff784eadeabb7b/01-pyspark-coalesce-wordcount.py)[**01-pyspark-coalesce-wordcount.py**](https://gist.github.com/dgadiraju/d6a920b806007dc49450293fae45beef#file-01-pyspark-coalesce-wordcount-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | lines = sc.textFile("/public/randomtextwriter/part-m-00000", 18) |
|  | words = lines.flatMap(lambda line: line.split(" ")) |
|  | wordTuples = words.map(lambda word: (word, 1)) |
|  | wc = wordTuples.reduceByKey(lambda x, y: x + y, 36) |
|  | wc.repartition(4).saveAsTextFile("/user/training/bootcamp/pyspark/wordcount03") |

[**view raw**](https://gist.github.com/dgadiraju/d6a920b806007dc49450293fae45beef/raw/7914a6a57e64092b0df2b3efbdff784eadeabb7b/02-pyspark-repartition-wordcount.py)[**02-pyspark-repartition-wordcount.py**](https://gist.github.com/dgadiraju/d6a920b806007dc49450293fae45beef#file-02-pyspark-repartition-wordcount-py) hosted with  by [**GitHub**](https://github.com/)

# Development and Deployment Life Cycle

posted on FEBRUARY 1, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

Now let us understand development life cycle using PyCharm and deployment lifecycle. As part of the deployment life cycle, we will see how to control runtime behavior.

## Development Life Cycle

Let us check the development life cycle of Spark applications using PyCharm with word count and daily revenue.

* Create a new project
* Make sure PyCharm is configured for Pyspark Python
* Externalize Properties using ConfigParser
* Create a Spark Configuration object and Spark Context object
* Develop logic to read, process and save the output back
* Externalize execution mode, input base directory, and output path
* Validate locally using pycharm and probably spark-submit in local mode

## Externalize Properties

Let us see how we can externalize properties to control the run time behavior based on the environment application is running.

## Deployment Life Cycle

Once the code is developed, we can deploy it on the gateway node on the cluster.

* Ship the folder which contains all python files to the gateway node
* Run using the spark-submit command
* Review capacity of the cluster.
  + Node Manager capacity
  + Default YARN Container -> Spark Executor configuration
  + Default Spark Executor configuration – 1 GB, 1 Core
* Here are the different modes in which we will be running and understand how it impact the execution (demo is done using word count).
  + Disable dynamic allocation and run with defaults (num-executors, executor-memory and executor-cores)
  + Understanding executor overhead and impact  on executor-memory
  + Increasing num-executors, executor-memory and executor-cores
  + Dynamic Allocation

GitHub repository for the code will be shared after session is done.

## Daily Revenue – Validate Locally

## Daily Revenue – Spark Configuration

## Word Count – Spark Configuration

[**KAIZEN**](http://wip.itversity.com/)

e-Learning Portal

* [**SHOP**](http://wip.itversity.com/)

* [**COURSES**](http://wip.itversity.com/courses/)

* [**BLOG**](http://wip.itversity.com/)

* [**FORUM**](http://discuss.itversity.com/)

* [**LOG IN**](http://wip.itversity.com/login/)

# Accumulators and Broadcast Variables

posted on FEBRUARY 1, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

Now let us understand about Accumulators and Broadcast Variables. They are also known as Shared Variables. Accumulators are primarily used as counters for sanity checks while broadcast variables are used for lookups. As part of this topic, we will also look into repartition and coalesce.

* Problem Statement
* High-level Design
* Validate Paths
* Accumulators
* Broadcast Variables
* Using broadcast

In this session, we will develop a program using HDFS APIs and then add accumulators to it.

You will not be able to see accumulator details as part of Spark UI for pyspark applications. However, you can read the variables after performing the action within the program. We have struggled a bit in the video, but the code is updated.

## Problem Statement

Before going into shared variables such as Accumulators and Broadcast Variables, let us define a new problem statement and come up with the solution. With this, we will understand how we can use HDFS APIs as part of applications built using Pyspark.

* We have to use orders, order\_items and products data set to compute revenue per product for a given month
* orders have order\_id and order\_date
* order\_items have order\_item\_subtotal, order\_item\_order\_id and order\_item\_product\_id
* products have product\_id and product\_name
* orders and order\_items are in HDFS and products data set is in the local file system

## High-level design

Let us come up with High level design before we go ahead and implement.

* Accept year and month as program argument (along with input path and output path)
* Filter for orders which fall in the month passed as an argument
* Join filtered orders and order\_items to get order\_item details for a given month
* Get revenue for each product\_id
* We need to read products from the local file system
* Convert into RDD and extract product\_id and product\_name
* Join it with aggregated order\_items (product\_id, revenue)
* Get product\_name and revenue for each product
* application.properties

|  |  |
| --- | --- |
|  | [dev] |
|  | executionMode = local |
|  |  |
|  | [prod] |
|  | executionMode = yarn-client |

[**view raw**](https://gist.github.com/dgadiraju/25f9ce83429b8c8410d714c7f25644c2/raw/2e076ae4c87d3f28704d0763f7c343fe7307ad02/pyspark-application.properties)[**pyspark-application.properties**](https://gist.github.com/dgadiraju/25f9ce83429b8c8410d714c7f25644c2#file-pyspark-application-properties) hosted with  by [**GitHub**](https://github.com/)

## Validate Paths

We might have to validate and/or manage both input paths and output paths before we trigger the execution.

* Input Path – We need to check whether the input path is valid or not.
* Output Path – We need to check if the directory already exists. If it does, we would like to delete and then run the applications.
* For validations as well as to manage directories we need to use HDFS APIs. As we are using Python as programming language we need to load Java Classes such as Path, FileSystem etc and use APIs .
* Create a new package by name retail
* Create a Python program with name RevenuePerProductForMonth
* src/main/python/retail/RevenuePerProductForMonth.py

|  |  |
| --- | --- |
|  | import sys |
|  | import ConfigParser as cp |
|  | try: |
|  | from pyspark import SparkConf, SparkContext |
|  | from pyspark.sql import SQLContext, Row, functions as func |
|  |  |
|  | props = cp.RawConfigParser() |
|  | props.read("src/main/resources/application.properties") |
|  |  |
|  | conf = SparkConf(). \ |
|  | setAppName("Total Revenue Per Day"). \ |
|  | setMaster(props.get(sys.argv[5], "executionMode")) |
|  |  |
|  | sc = SparkContext(conf=conf) |
|  | inputPath = sys.argv[1] |
|  | outputPath = sys.argv[2] |
|  | month = sys.argv[3] |
|  |  |
|  | Path = sc.\_gateway.jvm.org.apache.hadoop.fs.Path |
|  | FileSystem = sc.\_gateway.jvm.org.apache.hadoop.fs.FileSystem |
|  | Configuration = sc.\_gateway.jvm.org.apache.hadoop.conf.Configuration |
|  |  |
|  | fs = FileSystem.get(Configuration()) |
|  |  |
|  | if(fs.exists(Path(inputPath)) == False): |
|  | print("Input path does not exists") |
|  | else: |
|  | if(fs.exists(Path(outputPath))): |
|  | fs.delete(Path(outputPath), True) |
|  |  |
|  | # Filter for orders which fall in the month passed as argument |
|  | orders = inputPath + "/orders" |
|  | ordersFiltered = sc.textFile(orders). \ |
|  | filter(lambda order: month in order.split(",")[1]). \ |
|  | map(lambda order: (int(order.split(",")[0]), 1)) |
|  |  |
|  | # Join filtered orders and order\_items to get order\_item details for a given month |
|  | # Get revenue for each product\_id |
|  |  |
|  | orderItems = inputPath + "/order\_items" |
|  | revenueByProductId = sc.textFile(orderItems). \ |
|  | map(lambda orderItem: |
|  | (int(orderItem.split(",")[1]), |
|  | (int(orderItem.split(",")[2]), float(orderItem.split(",")[4]) |
|  | )) |
|  | ). \ |
|  | join(ordersFiltered). \ |
|  | map(lambda rec: rec[1][0]). \ |
|  | reduceByKey(lambda total, ele: total + ele) |
|  |  |
|  | # We need to read products from local file system |
|  | localPath = sys.argv[4] |
|  | productsFile = open(localPath + "/products/part-00000") |
|  | products = productsFile.read().splitlines() |
|  |  |
|  | # Convert into RDD and extract product\_id and product\_name |
|  | # Join it with aggregated order\_items (product\_id, revenue) |
|  | # Get product\_name and revenue for each product |
|  | sc.parallelize(products). \ |
|  | map(lambda product: |
|  | (int(product.split(",")[0]), product.split(",")[2])). \ |
|  | join(revenueByProductId). \ |
|  | map(lambda rec: rec[1][0] + "\t" + str(rec[1][1])). \ |
|  | saveAsTextFile(outputPath) |
|  |  |
|  | print ("Successfully imported Spark Modules") |
|  |  |
|  | except ImportError as e: |
|  | print ("Can not import Spark Modules", e) |
|  | sys.exit(1) |

[**view raw**](https://gist.github.com/dgadiraju/47ee1dadfe394865417b0544f6980bb7/raw/7f90fd9a3ab6f8be16690b76cc3d07fa3ceae05f/pyspark-RevenuePerProductForMonth.py)[**pyspark-RevenuePerProductForMonth.py**](https://gist.github.com/dgadiraju/47ee1dadfe394865417b0544f6980bb7#file-pyspark-revenueperproductformonth-py) hosted with  by [**GitHub**](https://github.com/)

* Run the spark job using spark-submit
* We can run the job using the following command.

|  |  |
| --- | --- |
|  | spark-submit \ |
|  | --master yarn \ |
|  | --conf spark.ui.port=54123 \ |
|  | src/main/python/retail/RevenuePerProductForMonthAccumulator.py \ |
|  | /public/retail\_db /user/dgadiraju/revenueperproductformonth \ |
|  | 2013-12 /data/retail\_db prod |

[**view raw**](https://gist.github.com/dgadiraju/81b329142549d3c1bca8ee60db2648de/raw/a9daf7c45285af65c4bd183552c433016f62ba50/spark-submit-python-revenueperproductformonthaccumulator.sh)[**spark-submit-python-revenueperproductformonthaccumulator.sh**](https://gist.github.com/dgadiraju/81b329142549d3c1bca8ee60db2648de#file-spark-submit-python-revenueperproductformonthaccumulator-sh) hosted with  by [**GitHub**](https://github.com/)

## Broadcast Variables

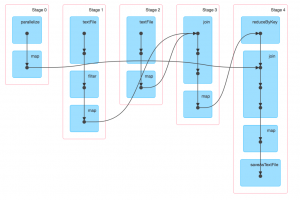
Broadcast Variable is another type of shared variable which can be broadcasted into all the executors and can access at runtime by tasks while processing data. It is typically used to replace joins with lookups when a very large data set is joined with small data set which can fit into the memory of executor JVM.

* At times we need to pass (broadcast) some information to all the executors
* It can be done by using broadcast variables
* A broadcast variable can be of a preliminary type or it could be a hash map
* Here are a few examples
  + Single value – Common discount percent for all the products
  + Hash map – look up or map-side join
* When very large data set (fact) is tried to join with smaller data set (dimension), broadcasting a dimension can have considerable performance improvement.
* Broadcast variables are immutable

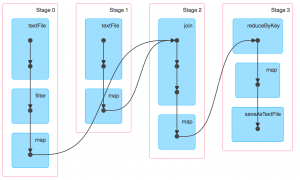
## Using broadcast

We can broadcast data set using broadcast. Let us see an example.

* We can read data from HDFS or local file system or even as configuration parameters
* Broadcast using the broadcast method of Spark Context
* Let us take the example of **Revenue per product for a given month**
* Earlier we have read **products** from the local file system, converted into RDD and then join with other RDD to get product name and revenue generated. Here is the DAG when the data is joined without using broadcast variables

[](https://kaizen.itversity.com/wp-content/uploads/2018/07/DAGRegularJoin.png)

* Here is how DAG look like after broadcasting products from the local file system. If we run this against a considerable amount of data, one can feel the difference in the performance because of broadcast variables

[](https://kaizen.itversity.com/wp-content/uploads/2018/07/DAGBroadCast.png)

* Here is the code for broadcast variables

|  |  |
| --- | --- |
|  | import sys |
|  | import ConfigParser as cp |
|  | try: |
|  | from pyspark import SparkConf, SparkContext |
|  | from pyspark.sql import SQLContext, Row, functions as func |
|  |  |
|  | props = cp.RawConfigParser() |
|  | props.read("src/main/resources/application.properties") |
|  |  |
|  | conf = SparkConf(). \ |
|  | setAppName("Total Revenue Per Day"). \ |
|  | setMaster(props.get(sys.argv[5], "executionMode")) |
|  |  |
|  | sc = SparkContext(conf=conf) |
|  | inputPath = sys.argv[1] |
|  | outputPath = sys.argv[2] |
|  | month = sys.argv[3] |
|  |  |
|  | Path = sc.\_gateway.jvm.org.apache.hadoop.fs.Path |
|  | FileSystem = sc.\_gateway.jvm.org.apache.hadoop.fs.FileSystem |
|  | Configuration = sc.\_gateway.jvm.org.apache.hadoop.conf.Configuration |
|  |  |
|  | fs = FileSystem.get(Configuration()) |
|  |  |
|  | if(fs.exists(Path(inputPath)) == False): |
|  | print("Input path does not exists") |
|  | else: |
|  | if(fs.exists(Path(outputPath))): |
|  | fs.delete(Path(outputPath), True) |
|  |  |
|  | # Filter for orders which fall in the month passed as argument |
|  | ordersCount = sc.accumulator(0) |
|  | orders = inputPath + "/orders" |
|  |  |
|  | def getOrdersTuples(rec): |
|  | ordersCount.add(1) |
|  | return (int(rec.split(",")[0]), 1) |
|  |  |
|  | ordersFiltered = sc.textFile(orders). \ |
|  | filter(lambda order: month in order.split(",")[1]). \ |
|  | map(getOrdersTuples) |
|  |  |
|  | # Join filtered orders and order\_items to get order\_item details for a given month |
|  | # Get revenue for each product\_id |
|  | orderItemsCount = sc.accumulator(0) |
|  | orderItems = inputPath + "/order\_items" |
|  |  |
|  | def getProductIdAndRevenue(rec): |
|  | orderItemsCount.add(1) |
|  | return rec[1][0] |
|  |  |
|  | revenueByProductId = sc.textFile(orderItems). \ |
|  | map(lambda orderItem: |
|  | (int(orderItem.split(",")[1]), |
|  | (int(orderItem.split(",")[2]), float(orderItem.split(",")[4]) |
|  | )) |
|  | ). \ |
|  | join(ordersFiltered). \ |
|  | map(getProductIdAndRevenue). \ |
|  | reduceByKey(lambda total, ele: total + ele) |
|  |  |
|  | # We need to read products from local file system |
|  | localPath = sys.argv[4] |
|  | productsFile = open(localPath + "/products/part-00000") |
|  | products = productsFile.read().splitlines() |
|  |  |
|  | # Extract product\_id and product\_name and create dict of it |
|  | # Broadcast the dict |
|  | productsDict = dict( |
|  | map(lambda product: |
|  | (int(product.split(",")[0]), product.split(",")[2]), products) |
|  | ) |
|  | bv = sc.broadcast(productsDict) |
|  |  |
|  | # Get product name for each product id in revenueByProductId |
|  | # by looking up in the broadcast variable |
|  |  |
|  | revenueByProductId. \ |
|  | map(lambda product: bv.value[product[0]] + "\t" + str(product[1])). \ |
|  | saveAsTextFile(outputPath) |
|  |  |
|  | except ImportError as e: |
|  | print ("Can not import Spark Modules", e) |
|  | sys.exit(1) |

[**view raw**](https://gist.github.com/dgadiraju/24d25775b41399b91d7c1c2a8eb3849c/raw/e5212d8ee74f7409bfb72c29fbce51e17ccfea0d/pyspark-RevenuePerProductForMonthBroadcast.py)[**pyspark-RevenuePerProductForMonthBroadcast.py**](https://gist.github.com/dgadiraju/24d25775b41399b91d7c1c2a8eb3849c#file-pyspark-revenueperproductformonthbroadcast-py) hosted with  by [**GitHub**](https://github.com/)

* Here is the code for submitting the job

|  |  |
| --- | --- |
|  | spark-submit \ |
|  | --master yarn \ |
|  | --conf spark.ui.port=54123 \ |
|  | src/main/python/retail/RevenuePerProductForMonthBroadcast.py \ |
|  | /public/retail\_db /user/dgadiraju/revenueperproductformonth \ |
|  | 2013-12 /data/retail\_db prod |

[**view raw**](https://gist.github.com/dgadiraju/75673500159e416fc527226a9d306dd9/raw/a9f4be529b44d5ec016b4aadf0ef2b96ea30239f/spark-submit-python-revenueperproductformonthbroadcast.sh%20)[**spark-submit-python-revenueperproductformonthbroadcast.sh**](https://gist.github.com/dgadiraju/75673500159e416fc527226a9d306dd9#file-spark-submit-python-revenueperproductformonthbroadcast-sh)hosted with  by [**GitHub**](https://github.com/)

# Creating Data Frames and Pre-Defined functions

posted on FEBRUARY 1, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

As part of this session, we will understand what is Data Frames, how data frames can be created from (text) files, hive tables, relational databases using JDBC etc. We will also understand how data frame can be registered as in-memory table/view and run SQL on top of it as well as some of the important functions that can be used to manipulate data as part of data frame operations.

* Data Frames – Overview
* Reading text data from files
* Reading data from hive
* Reading data from MySQL over JDBC
* Data Frame Operations – Overview
* Spark SQL – Overview
* Functions to manipulate data

## Data Frames – Overview

Data Frames is nothing but RDD with structure.

* Data Frame can be created on any data set which has structure associated with it.
* The attributes/columns in a data frame can be referred using names.
* One can create a data frame using data from files, hive tables, relational tables over JDBC.
* Common functions on Data Frames
  + printSchema – to print the column names and data types of the data frame
  + show – to preview data (default 20 records)
  + describe – to understand the characteristics of data
  + count – to get number of records
  + collect – to convert data frame into Array
* Once the data frame is created, we can process data using 2 approaches.
  + Native Data Frame APIs
  + Register as temp table and run queries using spark.sql
* To work with Data Frames as well as Spark SQL, we need to create an object of type SparkSession

|  |  |
| --- | --- |
|  | from pyspark.sql import SparkSession |
|  |  |
|  | spark = SparkSession. \ |
|  | builder. \ |
|  | master('local'). \ |
|  | appName('Create Dataframe over JDBC'). \ |
|  | getOrCreate() |

[**view raw**](https://gist.github.com/dgadiraju/a50ad8b011b4625a8bc61c0d04964468/raw/d4e7ebd63bbeaa4b4dbcc9daf814f9597f97f36b/pyspark-create-spark.py)[**pyspark-create-spark.py**](https://gist.github.com/dgadiraju/a50ad8b011b4625a8bc61c0d04964468#file-pyspark-create-spark-py) hosted with  by [**GitHub**](https://github.com/)

* Once the SparkSession object is created we can use APIs under spark.read to create data frame or use spark.sql to run queries on hive tables or temp tables.

## Reading text data from files

Let us see how we can read text data from files into a data frame. spark.read also have APIs for other types of file formats, but we will get into those details later.

* We can use spark.read.csv or spark.read.text to read text data.
* spark.read.csv can be used for comma separated data. Default field names will be in the form of \_c0, \_c1 etc. We can pass the delimiter using the keyword argument using sep to spark.read.csv.
* We can also use spark.read.format with the file type. We can use schema to define schema, option such as sep to pass delimiter and load to load data from a given location into Data Frame.
* spark.read.text can be used to read fixed length data where there is no delimiter. Default field name is value.
* We can also define attribute names using the toDF function
* In either of the case data will be represented as strings
* We can convert data types by using cast function – df.select(df.field.cast(IntegerType()))
* We will see all other functions soon, but let us perform the task of reading the data into the data frame and represent it in their original format.

|  |  |
| --- | --- |
|  | spark = SparkSession. \ |
|  | builder. \ |
|  | master('local'). \ |
|  | appName('CSV Example'). \ |
|  | getOrCreate() |
|  |  |
|  | orders = spark.read. \ |
|  | format('csv'). \ |
|  | schema('order\_id int, order\_date string, order\_customer\_id int, order\_status string'). \ |
|  | load('/Users/itversity/Research/data/retail\_db/orders') |
|  |  |
|  | orders.printSchema() |
|  | orders.show() |
|  |  |
|  | orderItems = spark.read. \ |
|  | format('csv'). \ |
|  | schema('''order\_item\_id int, |
|  | order\_item\_order\_id int, |
|  | order\_item\_product\_id int, |
|  | order\_item\_quantity int, |
|  | order\_item\_subtotal float, |
|  | order\_item\_product\_price float |
|  | '''). \ |
|  | load('/Users/itversity/Research/data/retail\_db/order\_items') |
|  |  |
|  | orderItems.printSchema() |
|  | orderItems.show() |

[**view raw**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7/raw/af963c48c5cb1a613ff7cb6fab3bd16b97fb1999/pyspark-dataframe-01-csv-example.py)[**pyspark-dataframe-01-csv-example.py**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7#file-pyspark-dataframe-01-csv-example-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # In case you are using pycharm, first you need to create object of type SparkSession |
|  | spark = SparkSession. \ |
|  | builder. \ |
|  | master('local'). \ |
|  | appName('CSV Example'). \ |
|  | getOrCreate() |
|  |  |
|  | ordersCSV = spark.read. \ |
|  | csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | orderItemsCSV = spark.read. \ |
|  | csv('/public/retail\_db/order\_items'). \ |
|  | toDF('order\_item\_id', 'order\_item\_order\_id', 'order\_item\_product\_id', |
|  | 'order\_item\_quantity', 'order\_item\_subtotal', 'order\_item\_product\_price') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  |  |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.printSchema() |
|  | orders.show() |
|  |  |
|  | orderItems = orderItemsCSV.\ |
|  | withColumn('order\_item\_id', orderItemsCSV.order\_item\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_order\_id', orderItemsCSV.order\_item\_order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_product\_id', orderItemsCSV.order\_item\_product\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_quantity', orderItemsCSV.order\_item\_quantity.cast(IntegerType())). \ |
|  | withColumn('order\_item\_subtotal', orderItemsCSV.order\_item\_subtotal.cast(FloatType())). \ |
|  | withColumn('order\_item\_product\_price', orderItemsCSV.order\_item\_product\_price.cast(FloatType())) |
|  |  |
|  | orderItems.printSchema() |
|  | orderItems.show() |

[**view raw**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7/raw/af963c48c5cb1a613ff7cb6fab3bd16b97fb1999/pyspark-dataframe-02-csv-example.py)[**pyspark-dataframe-02-csv-example.py**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7#file-pyspark-dataframe-02-csv-example-py) hosted with  by [**GitHub**](https://github.com/)

## Reading data from hive

If Hive and Spark are integrated, we can create data frames from data in Hive tables or run Spark SQL queries against it.

* We can use spark.read.table to read data from Hive tables into Data Frame
* We can prefix database name to table name while reading Hive tables into Data Frame
* We can also run Hive queries directly using spark.sql
* Both spark.read.table and spark.sql returns Data Frame

## Reading data from MySQL over JDBC

Spark also facilitate us to read data from relational databases over JDBC.

* We need to make sure JDBC jar file is registered using --packages or --jars and --driver-class-path while launching pyspark
* In Pycharm, we need to copy the relevant JDBC jar file to SPARK\_HOME/jars
* We can either use spark.read.format(‘jdbc’) with options or spark.read.jdbc with jdbc URL, table name and other properties as dict to read data from remote relational databases.
* We can pass a table name or query to read data using JDBC into Data Frame
* While reading data, we can define a number of partitions (using numPartitions), criteria to divide data into partitions (partitionColumn, lowerBound, upperBound)
* Partitioning can be done only on numeric fields
* If lowerBound and upperBound are specified, it will generate strides depending upon the number of partitions and then process entire data. Here is the example
  + We are trying to read order\_items data with 4 as numPartitions
  + partitionColumn – order\_item\_order\_id
  + lowerBound – 10000
  + upperBound – 20000
  + order\_item\_order\_id is in the range of 1 and 68883
  + But as we define lowerBound as 10000 and upperBound as 20000, there will be strides – 1 to 12499, 12500 to 14999, 15000 to 17499, 17500 to a maximum of order\_item\_order\_id
  + You can check the data in the output path mentioned

|  |  |
| --- | --- |
|  | from pyspark.sql import SparkSession |
|  |  |
|  | spark = SparkSession. \ |
|  | builder. \ |
|  | master('local'). \ |
|  | appName('Create Dataframe over JDBC'). \ |
|  | getOrCreate() |
|  |  |
|  | orders = spark.read. \ |
|  | format('jdbc'). \ |
|  | option('url', 'jdbc:mysql://ms.itversity.com'). \ |
|  | option('dbtable', 'retail\_db.orders'). \ |
|  | option('user', 'retail\_user'). \ |
|  | option('password', 'itversity'). \ |
|  | load() |
|  |  |
|  | orders.show() |
|  |  |
|  | orderItems = spark.read. \ |
|  | jdbc("jdbc:mysql://ms.itversity.com", "retail\_db.order\_items", |
|  | properties={"user": "retail\_user", |
|  | "password": "itversity", |
|  | "numPartitions": "4", |
|  | "partitionColumn": "order\_item\_order\_id", |
|  | "lowerBound": "10000", |
|  | "upperBound": "20000"}) |
|  |  |
|  | orderItems.write.json('/user/training/bootcamp/pyspark/orderItemsJDBC') |
|  |  |
|  | query = "(select order\_status, count(1) from retail\_db.orders group by order\_status) t" |
|  | queryData = spark.read. \ |
|  | jdbc("jdbc:mysql://ms.itversity.com", query, |
|  | properties={"user": "retail\_user", |
|  | "password": "itversity"}) |
|  |  |
|  | queryData.show() |

[**view raw**](https://gist.github.com/dgadiraju/7817fdd69bf8b4f15fe3225c5d8897bf/raw/22e8abfdf1515ff64f5790dc5995e6e2f53faade/pyspark-create-dataframe-jdbc.py)[**pyspark-create-dataframe-jdbc.py**](https://gist.github.com/dgadiraju/7817fdd69bf8b4f15fe3225c5d8897bf#file-pyspark-create-dataframe-jdbc-py) hosted with  by [**GitHub**](https://github.com/)

## Data Frame Operations – Overview

Let us see an overview of Data Frame Operations. We can also process Data Frames using Spark SQL.

* Selection or Projection – select
* Filtering data – filter or where
* Joins – join (supports outer join as well)
* Aggregations – groupBy and agg with the support of functions such as sum, avg, min, max etc
* Sorting – sort or orderBy
* Analytics Functions – aggregations, ranking and windowing functions

## Spark SQL – Overview

We can also use Spark SQL to process data in data frames.

* We can get a list of tables by using spark.sql('show tables')
* We can register the data frame as a temporary view df.createTempView("view\_name")
* The output of show tables show the temporary tables as well
* Once temp view is created, we can use SQL style syntax and run queries against the tables/views
* Most of the hive queries will work out of the box

## Functions to manipulate data

Let us quickly look into some of the functions available in Data Frames.

* Main package for functions pyspark.sql.functions
* We can import by saying from pyspark.sql import functions as sf
* You will see many functions which are similar to the functions in traditional databases.
* These can be categorized into
  + String manipulation
  + Date manipulation
  + Typecasting
  + Expressions such as the case when
* We will see some of the functions in action
  + substring
  + lower, upper
  + trim
  + date\_format
  + trunc
  + Type Casting
  + case when

# Data Frame Operations – Basic Transformations

posted on FEBRUARY 3, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

As part of this session, we will see basic transformations we can perform on top of Data Frames such as filtering, aggregations, joins etc. We will build an end to end application by taking a simple problem statement.

* Data Frame Operations – APIs
* Problem Statement – Get daily product revenue
* Projecting data using select, withColumn and selectExpr
* Filtering data using where or filter
* Joining Data Sets
* Grouping data and performing aggregations
* Sorting data
* Development Life Cycle

## Data Frame Operations – APIs

Let us recap about Data Frame Operations. It is one of the 2 ways we can process Data Frames.

* Selection or Projection – select
* Filtering data – filter or where
* Joins – join (supports outer join as well)
* Aggregations – groupBy and agg with the support of functions such as sum, avg, min, max etc
* Sorting – sort or orderBy
* Analytics Functions – aggregations, ranking and windowing functions

## Problem Statement – Get daily product revenue

Here is the problem statement for which we will be exploring Data Frame APIs to come up with the final solution.

* Get daily product revenue
* orders – order\_id, order\_date, order\_customer\_id, order\_status
* order\_items – order\_item\_id, order\_item\_order\_id, order\_item\_product\_id, order\_item\_quantity, order\_item\_subtotal, order\_item\_product\_price
* Data is comma separated
* We will fetch data using spark.read.csv
* Apply type cast functions to convert fields into their original type where ever is applicable.

|  |  |
| --- | --- |
|  | spark = SparkSession. \ |
|  | builder. \ |
|  | master('local'). \ |
|  | appName('CSV Example'). \ |
|  | getOrCreate() |
|  |  |
|  | orders = spark.read. \ |
|  | format('csv'). \ |
|  | schema('order\_id int, order\_date string, order\_customer\_id int, order\_status string'). \ |
|  | load('/Users/itversity/Research/data/retail\_db/orders') |
|  |  |
|  | orders.printSchema() |
|  | orders.show() |
|  |  |
|  | orderItems = spark.read. \ |
|  | format('csv'). \ |
|  | schema('''order\_item\_id int, |
|  | order\_item\_order\_id int, |
|  | order\_item\_product\_id int, |
|  | order\_item\_quantity int, |
|  | order\_item\_subtotal float, |
|  | order\_item\_product\_price float |
|  | '''). \ |
|  | load('/Users/itversity/Research/data/retail\_db/order\_items') |
|  |  |
|  | orderItems.printSchema() |
|  | orderItems.show() |

[**view raw**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7/raw/af963c48c5cb1a613ff7cb6fab3bd16b97fb1999/pyspark-dataframe-01-csv-example.py)[**pyspark-dataframe-01-csv-example.py**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7#file-pyspark-dataframe-01-csv-example-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # In case you are using pycharm, first you need to create object of type SparkSession |
|  | spark = SparkSession. \ |
|  | builder. \ |
|  | master('local'). \ |
|  | appName('CSV Example'). \ |
|  | getOrCreate() |
|  |  |
|  | ordersCSV = spark.read. \ |
|  | csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | orderItemsCSV = spark.read. \ |
|  | csv('/public/retail\_db/order\_items'). \ |
|  | toDF('order\_item\_id', 'order\_item\_order\_id', 'order\_item\_product\_id', |
|  | 'order\_item\_quantity', 'order\_item\_subtotal', 'order\_item\_product\_price') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  |  |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.printSchema() |
|  | orders.show() |
|  |  |
|  | orderItems = orderItemsCSV.\ |
|  | withColumn('order\_item\_id', orderItemsCSV.order\_item\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_order\_id', orderItemsCSV.order\_item\_order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_product\_id', orderItemsCSV.order\_item\_product\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_quantity', orderItemsCSV.order\_item\_quantity.cast(IntegerType())). \ |
|  | withColumn('order\_item\_subtotal', orderItemsCSV.order\_item\_subtotal.cast(FloatType())). \ |
|  | withColumn('order\_item\_product\_price', orderItemsCSV.order\_item\_product\_price.cast(FloatType())) |
|  |  |
|  | orderItems.printSchema() |
|  | orderItems.show() |

[**view raw**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7/raw/af963c48c5cb1a613ff7cb6fab3bd16b97fb1999/pyspark-dataframe-02-csv-example.py)[**pyspark-dataframe-02-csv-example.py**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7#file-pyspark-dataframe-02-csv-example-py) hosted with  by [**GitHub**](https://github.com/)

## Projecting or Selecting Data

Now let us see how we can project data the way we want from Data Frames using Data Frame APIs. We can either use select, withColumn or selectExpr to project the data.

* We can use select and fetch data from the fields we are looking for.
* Both orders and orderItems are of type DataFrames. We will be able to access attributes by prefixing data frame name (e. g.: orders.order\_id and orderItems.order\_item\_id). Also, we can pass attribute names as strings.
* e.g.: orders.select(orders.order\_id, orders.order\_date) and orders.select('order\_id', 'order\_date')
* We can apply necessary functions to manipulate data while it is being projected – orders.select(substring('order\_date', 1, 7)).show()
* We can give aliases to the derived fields using alias function – orders.select(substring('order\_date', 1, 7).alias('order\_month')).show()
* If we want to add new field derived from existing fields we can use the withColumn function. The first argument is an alias and 2nd argument is data processing logic – orders.withColumn('order\_month', substring('order\_date', 1, 7).alias('order\_month')).show()
* If the alias is same as existing field, then the field in new data frame will contain transformed data.

## Filtering data using where or filter

Data Frame has 2 APIs to filter the data, where and filter. They are just synonyms and you can use either of them for filtering.

* You can use a filter or where using 2 overloaded functions. One takes SQL style syntax and other takes Data Frame Native style syntax.
* One by using class.attributeName and comparing with values – e. g.: orders.where(orders.order\_status == 'COMPLETE').show()
* Other by passing conditions as literals – e. g.: orders.where('order\_status = "COMPLETE"').show()
* Make sure both orders and orderItems data frames are created
* Let us see a few more examples
  + Get orders which are either COMPLETE or CLOSED
  + Get orders which are either COMPLETE or CLOSED and placed in the month of 2013 August
  + Get order items where order\_item\_subtotal is not equal to the product of order\_item\_quantity and order\_item\_product\_price
  + Get all the orders which are placed on first of every month

|  |  |
| --- | --- |
|  | # Get orders which are either COMPLETE or CLOSED |
|  | orders.where('order\_status = "COMPLETE" or order\_status = "CLOSED"').show() |
|  | orders.where('order\_status in ("COMPLETE", "CLOSED")').show() |
|  | orders.where((orders.order\_status == 'COMPLETE') | (orders.order\_status == 'CLOSED')).show() |
|  | orders.where((orders.order\_status == 'COMPLETE').\_\_or\_\_(orders.order\_status == 'CLOSED')).show() |
|  | orders.where(orders.order\_status.isin('COMPLETE', 'CLOSED')).show() |

[**view raw**](https://gist.github.com/dgadiraju/b04db12e89f164fc1a0e5aa1db04c4c4/raw/cd24129aee6618fedb7327cfefcad6073a389b80/pyspark-dataframes-filtering-01.py)[**pyspark-dataframes-filtering-01.py**](https://gist.github.com/dgadiraju/b04db12e89f164fc1a0e5aa1db04c4c4#file-pyspark-dataframes-filtering-01-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get orders which are either COMPLETE or CLOSED and placed in month of 2013 August |
|  |  |
|  | orders.where('order\_status in ("COMPLETE", "CLOSED") and order\_date like "2013-08%"').show() |
|  | orders.where(orders.order\_status.isin('COMPLETE', 'CLOSED').\_\_and\_\_(orders.order\_date.like('2013-08%'))).show() |

[**view raw**](https://gist.github.com/dgadiraju/b04db12e89f164fc1a0e5aa1db04c4c4/raw/cd24129aee6618fedb7327cfefcad6073a389b80/pyspark-dataframes-filtering-02.py)[**pyspark-dataframes-filtering-02.py**](https://gist.github.com/dgadiraju/b04db12e89f164fc1a0e5aa1db04c4c4#file-pyspark-dataframes-filtering-02-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get order items where order\_item\_subtotal is not equal to product of order\_item\_quantity and order\_item\_product\_price |
|  | orderItems.where('order\_item\_subtotal != round(order\_item\_quantity \* order\_item\_product\_price, 2)').show() |
|  |  |
|  | from pyspark.sql.functions import round |
|  | orderItems.where(orderItems.order\_item\_subtotal != |
|  | round((orderItems.order\_item\_quantity \* orderItems.order\_item\_product\_price), 2) |
|  | ).show() |

[**view raw**](https://gist.github.com/dgadiraju/b04db12e89f164fc1a0e5aa1db04c4c4/raw/cd24129aee6618fedb7327cfefcad6073a389b80/pyspark-dataframes-filtering-03.py)[**pyspark-dataframes-filtering-03.py**](https://gist.github.com/dgadiraju/b04db12e89f164fc1a0e5aa1db04c4c4#file-pyspark-dataframes-filtering-03-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get all the orders which are placed on first of every month |
|  | orders.where('date\_format(order\_date, "dd") = "01"').show() |
|  |  |
|  | from pyspark.sql.functions import date\_format |
|  | orders.where(date\_format(orders.order\_date, 'dd') == '01').show() |

[**view raw**](https://gist.github.com/dgadiraju/b04db12e89f164fc1a0e5aa1db04c4c4/raw/cd24129aee6618fedb7327cfefcad6073a389b80/pyspark-dataframes-filtering-04.py)[**pyspark-dataframes-filtering-04.py**](https://gist.github.com/dgadiraju/b04db12e89f164fc1a0e5aa1db04c4c4#file-pyspark-dataframes-filtering-04-py) hosted with  by [**GitHub**](https://github.com/)

## Joining Data Sets

Quite often we need to deal with multiple data sets which are related to each other.

* We need to first understand the relationship with respect to data sets
* All our data sets have relationships defined between them.
  + orders and order\_items are transaction tables. orders is parent and order\_items is a child. The relationship is established between the two using order\_id (in order\_items, it is represented as order\_item\_order\_id)
  + We also have product catalog normalized into 3 tables – products, categories, and departments (with relationships established in that order)
  + We also have a customers table
  + There is a relationship between customers and orders – customers is parent data set as one customer can place multiple orders.
  + There is a relationship between the product catalog and order\_items via products – products is parent data set as one product can be ordered as part of multiple order\_items.
* Determine the type of join – inner or outer (left or right or full)
* Data Frames have an API called join to perform joins
* We can make the join outer by passing an additional argument
* Let us see few examples
  + Get all the order items corresponding to COMPLETE or CLOSED orders
  + Get all the orders where there are no corresponding order\_items
  + Check if there are any order\_items where there is no corresponding order in the orders data set

|  |  |
| --- | --- |
|  | # Get all the order items corresponding to COMPLETE or CLOSED orders |
|  |  |
|  | orders.where('order\_status in ("COMPLETE", "CLOSED")'). \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/1f5bd6c53fedc26318501d7646e202ae/raw/d1f612c4c3585d94fc641a3ecfbc320f6360186d/pyspark-dataframes-join-01.py)[**pyspark-dataframes-join-01.py**](https://gist.github.com/dgadiraju/1f5bd6c53fedc26318501d7646e202ae#file-pyspark-dataframes-join-01-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get all the orders where there are no corresponding order\_items |
|  |  |
|  | orders. \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id, 'left'). \ |
|  | where('order\_item\_order\_id is null'). \ |
|  | select('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status'). \ |
|  | show() |
|  |  |
|  | orders. \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id, 'left'). \ |
|  | where(orderItems.order\_item\_order\_id.isNull()). \ |
|  | select(orders.order\_id, orders.order\_date, orders.order\_customer\_id, orders.order\_status). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/1f5bd6c53fedc26318501d7646e202ae/raw/d1f612c4c3585d94fc641a3ecfbc320f6360186d/pyspark-dataframes-join-02.py)[**pyspark-dataframes-join-02.py**](https://gist.github.com/dgadiraju/1f5bd6c53fedc26318501d7646e202ae#file-pyspark-dataframes-join-02-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Check if there are any order\_items where there is no corresponding order in orders data set |
|  |  |
|  | orders. \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id, 'right'). \ |
|  | where('order\_id is null'). \ |
|  | select('order\_item\_id', 'order\_item\_order\_id'). \ |
|  | show() |
|  |  |
|  | orders. \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id, 'right'). \ |
|  | where(orders.order\_id.isNull()). \ |
|  | select(orderItems.order\_item\_id, orderItems.order\_item\_order\_id). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/1f5bd6c53fedc26318501d7646e202ae/raw/d1f612c4c3585d94fc641a3ecfbc320f6360186d/pyspark-dataframes-join-03.py)[**pyspark-dataframes-join-03.py**](https://gist.github.com/dgadiraju/1f5bd6c53fedc26318501d7646e202ae#file-pyspark-dataframes-join-03-py) hosted with  by [**GitHub**](https://github.com/)

## Grouping data and performing aggregations

Many times we want to perform aggregations such as sum, average, minimum, maximum etc within each group. We need to first group the data and then perform aggregation.

* groupBy is the function which can be used to group the data on one or more columns
* Once data is grouped we can perform all supported aggregations – sum, avg, min, max etc
* We can invoke the functions directly or as part of agg
* agg gives us more flexibility to give aliases to the derived fields
* Let us see few examples
  + Get count by status from orders
  + Get revenue for each order id from order items
  + Get daily product revenue (order\_date and order\_item\_product\_id are part of keys, order\_item\_subtotal is used for aggregation)

|  |  |
| --- | --- |
|  | # Get count by status from orders |
|  | orders.groupBy('order\_status').count().show() |
|  | orders.groupBy('order\_status'). \ |
|  | agg(count('order\_status').alias('status\_count')). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/feebadf5c89d96fa4aab87651066cc51/raw/00aeb1de99eb477134098b3981900386339ee7d3/pyspark-dataframes-group-and-agg-01.py)[**pyspark-dataframes-group-and-agg-01.py**](https://gist.github.com/dgadiraju/feebadf5c89d96fa4aab87651066cc51#file-pyspark-dataframes-group-and-agg-01-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get revenue for each order id from order items |
|  | orderItems.groupBy('order\_item\_order\_id'). \ |
|  | sum('order\_item\_subtotal'). \ |
|  | show() |
|  |  |
|  | from pyspark.sql.functions import round, sum |
|  | orderItems.groupBy('order\_item\_order\_id'). \ |
|  | agg(round(sum('order\_item\_subtotal'), 2).alias('order\_revenue')). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/feebadf5c89d96fa4aab87651066cc51/raw/00aeb1de99eb477134098b3981900386339ee7d3/pyspark-dataframes-group-and-agg-02.py)[**pyspark-dataframes-group-and-agg-02.py**](https://gist.github.com/dgadiraju/feebadf5c89d96fa4aab87651066cc51#file-pyspark-dataframes-group-and-agg-02-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get daily product revenue |
|  | # filter for complete and closed orders |
|  | # groupBy order\_date and order\_item\_product\_id |
|  | # Use agg and sum on order\_item\_subtotal to get revenue |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | from pyspark.sql.functions import sum, round |
|  | orders.where('order\_status in ("COMPLETE", "CLOSED")'). \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id). \ |
|  | groupBy('order\_date', 'order\_item\_product\_id'). \ |
|  | agg(round(sum('order\_item\_subtotal'), 2).alias('revenue')). \ |
|  | show() |
|  |  |
|  | orders.where('order\_status in ("COMPLETE", "CLOSED")'). \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id). \ |
|  | groupBy(orders.order\_date, orderItems.order\_item\_product\_id). \ |
|  | agg(round(sum(orderItems.order\_item\_subtotal), 2).alias('revenue')). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/feebadf5c89d96fa4aab87651066cc51/raw/00aeb1de99eb477134098b3981900386339ee7d3/pyspark-dataframes-group-and-agg-03.py)[**pyspark-dataframes-group-and-agg-03.py**](https://gist.github.com/dgadiraju/feebadf5c89d96fa4aab87651066cc51#file-pyspark-dataframes-group-and-agg-03-py) hosted with  by [**GitHub**](https://github.com/)

## Sorting data

Now let us see how we can sort the data using sort or orderBy.

* sort or orderBy can be used to sort the data globally.
* We can perform composite sorting by using multiple fields
* By default, data will be sorted in ascending order
* We can change the order by using desc function.
* At times, we might not want to sort the data globally. Instead, we might want to sort the data within a group. In that case, we can use sortWithinPartitions (e. g.: Sort the stores by revenue within each state)
* Let us see few examples
  + Sort orders by status
  + Sort orders by date and then by status
  + Sort order items by order\_item\_order\_id and order\_item\_subtotal descending
  + Take daily product revenue data and sort in ascending order by date and then descending order by revenue.

|  |  |
| --- | --- |
|  | # Sort orders by status |
|  | orders.sort('order\_status').show() |
|  | orders.orderBy('order\_status').show() |
|  | orders.orderBy(orders.order\_status).show() |

[**view raw**](https://gist.github.com/dgadiraju/7021dabdae1d0a434dc08c2734206ed0/raw/cbb46f1d5dff597c553e8cf017f1070a0e0c79e4/pyspark-dataframes-sorting-01.py)[**pyspark-dataframes-sorting-01.py**](https://gist.github.com/dgadiraju/7021dabdae1d0a434dc08c2734206ed0#file-pyspark-dataframes-sorting-01-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | #Sort orders by date and then by status |
|  | orders.sort('order\_date', 'order\_status').show() |
|  | orders.orderBy(orders.order\_date, orders.order\_status).show() |

[**view raw**](https://gist.github.com/dgadiraju/7021dabdae1d0a434dc08c2734206ed0/raw/cbb46f1d5dff597c553e8cf017f1070a0e0c79e4/pyspark-dataframes-sorting-02.py)[**pyspark-dataframes-sorting-02.py**](https://gist.github.com/dgadiraju/7021dabdae1d0a434dc08c2734206ed0#file-pyspark-dataframes-sorting-02-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Sort order items by order\_item\_order\_id and order\_item\_subtotal descending |
|  | orderItems. \ |
|  | sort(orderItems.order\_item\_order\_id, orderItems.order\_item\_subtotal.desc()). \ |
|  | show() |
|  |  |
|  | orderItems. \ |
|  | orderBy(orderItems.order\_item\_order\_id, orderItems.order\_item\_subtotal.desc()). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/7021dabdae1d0a434dc08c2734206ed0/raw/cbb46f1d5dff597c553e8cf017f1070a0e0c79e4/pyspark-dataframes-sorting-03.py)[**pyspark-dataframes-sorting-03.py**](https://gist.github.com/dgadiraju/7021dabdae1d0a434dc08c2734206ed0#file-pyspark-dataframes-sorting-03-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Take daily product revenue data and |
|  | # sort in ascending order by date and |
|  | # then descending order by revenue. |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | from pyspark.sql.functions import sum, round |
|  |  |
|  | dailyProductRevenue = orders.where('order\_status in ("COMPLETE", "CLOSED")'). \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id). \ |
|  | groupBy(orders.order\_date, orderItems.order\_item\_product\_id). \ |
|  | agg(round(sum(orderItems.order\_item\_subtotal), 2).alias('revenue')) |
|  |  |
|  | dailyProductRevenueSorted = dailyProductRevenue. \ |
|  | orderBy(dailyProductRevenue.order\_date, dailyProductRevenue.revenue.desc()) |
|  |  |
|  | dailyProductRevenueSorted.show() |

[**view raw**](https://gist.github.com/dgadiraju/7021dabdae1d0a434dc08c2734206ed0/raw/cbb46f1d5dff597c553e8cf017f1070a0e0c79e4/pyspark-dataframes-sorting-04.py)[**pyspark-dataframes-sorting-04.py**](https://gist.github.com/dgadiraju/7021dabdae1d0a434dc08c2734206ed0#file-pyspark-dataframes-sorting-04-py) hosted with  by [**GitHub**](https://github.com/)

## Development Life Cycle

Let us develop the application using Pycharm and run it on the cluster.

* Make sure application.properties have required input path and output path along with execution mode
* Read orders and order\_items data into data frames
* Filter for complete and closed orders
* Join with order\_items
* Aggregate to get revenue for each order\_date and order\_item\_product\_id
* Sort in ascending order by date and then descending order by revenue
* Save the output as CSV format
* Validate using Pycharm

#### Run on the Cluster

As we have developed and validated using PyCharm, now it is time for us to validate the application using spark-submit both locally as well as on the cluster.

* Validate using spark-submit locally
* Ship it to the gateway node of the cluster
* Run on the gateway node of the cluster
* Validate the output directory

|  |  |
| --- | --- |
|  | [dev] |
|  | executionMode = local |
|  | input.base.dir = /Users/itversity/Research/data/retail\_db |
|  | output.base.dir = /Users/itversity/Research/data/bootcamp/pyspark |
|  |  |
|  | [prod] |
|  | executionMode = yarn-client |
|  | input.base.dir = /public/retail\_db |
|  | output.base.dir = /user/training/bootcamp/pyspark |

[**view raw**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58/raw/d7768968719340ccc67ba6ef639330fe42ae097d/pyspark-dataframes-01-application.properties)[**pyspark-dataframes-01-application.properties**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58#file-pyspark-dataframes-01-application-properties) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | import configparser as cp, sys |
|  | from pyspark.sql import SparkSession |
|  |  |
|  | props = cp.RawConfigParser() |
|  | props.read('src/main/resources/application.properties') |
|  | env = sys.argv[1] |
|  |  |
|  | spark = SparkSession.\ |
|  | builder.\ |
|  | appName('Daily Product Revenue using Data Frame Operations').\ |
|  | master(props.get(env, 'executionMode')).\ |
|  | getOrCreate() |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | inputBaseDir = props.get(env, 'input.base.dir') |
|  | orders = spark.read. \ |
|  | format('csv'). \ |
|  | schema('''order\_id INT, |
|  | order\_date STRING, |
|  | order\_customer\_id INT, |
|  | order\_status STRING |
|  | '''). \ |
|  | load(inputBaseDir + '/orders') |
|  |  |
|  | orderItems = spark.read. \ |
|  | format('csv'). \ |
|  | schema('''order\_item\_id INT, |
|  | order\_item\_order\_id INT, |
|  | order\_item\_product\_id INT, |
|  | order\_item\_quantity INT, |
|  | order\_item\_subtotal FLOAT, |
|  | order\_item\_product\_price FLOAT |
|  | '''). \ |
|  | load(inputBaseDir + '/order\_items') |
|  |  |
|  | from pyspark.sql.functions import sum, round |
|  | dailyProductRevenue = orders. \ |
|  | where('order\_status in ("COMPLETE", "CLOSED")'). \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id). \ |
|  | groupBy('order\_date', 'order\_item\_product\_id'). \ |
|  | agg(round(sum(orderItems.order\_item\_subtotal), 2).alias('revenue')) |
|  |  |
|  | dailyProductRevenueSorted = dailyProductRevenue. \ |
|  | orderBy(dailyProductRevenue.order\_date, dailyProductRevenue.revenue.desc()) |
|  |  |
|  | outputBaseDir = props.get(env, 'output.base.dir') |
|  | dailyProductRevenueSorted.write.csv(outputBaseDir + '/daily\_product\_revenue') |

[**view raw**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58/raw/d7768968719340ccc67ba6ef639330fe42ae097d/pyspark-dataframes-02-daily-product-revenue.py)[**pyspark-dataframes-02-daily-product-revenue.py**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58#file-pyspark-dataframes-02-daily-product-revenue-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | spark-submit \ |
|  | --master yarn \ |
|  | --deploy-mode client \ |
|  | --conf spark.ui.port=12901 \ |
|  | src/main/python/retail\_db/df/DailyProductRevenueDFO.py \ |
|  | prod |

[**view raw**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58/raw/d7768968719340ccc67ba6ef639330fe42ae097d/pyspark-dataframes-03-daily-product-revenue.sh)[**pyspark-dataframes-03-daily-product-revenue.sh**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58#file-pyspark-dataframes-03-daily-product-revenue-sh) hosted with  by [**GitHub**](https://github.com/)

## Exercises

Try to develop programs for these exercises

* Get number of closed or complete orders placed by each customer
* Get revenue generated by each customer for the month of 2014 January (consider only closed or complete orders)
* Get revenue generated by each product on a monthly basis – get product name, month and revenue generated by each product (round off revenue to 2 decimals)
* Get revenue generated by each product category on a daily basis – get category name, date and revenue generated by each category (round off revenue to 2 decimals)
* Get the details of the customers who never placed orders

# Data Frame Operations – Analytics or Windowing Functions

posted on FEBRUARY 3, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

As part of this session, we will see advanced operations such as aggregations, ranking, and windowing functions within each group using APIs such as over, partitionBy etc. We will also build a solution to the problem and run it on a multinode cluster.

* Window Functions – APIs
* Problem Statement
* Creating Window Spec
* Performing Aggregations
* Using Windowing Functions
* Ranking Functions
* Development Life Cycle

## Window Functions – APIs

Let us understand APIs related to aggregations, ranking and windowing functions.

* Main package **pyspark.sql.window**
* It has classes such as **Window** and **WindowSpec**
* Window have APIs such as **partitionBy**, **orderBy** etc
* These APIs (such as partitionBy) return **WindowSpec** object. We can pass WindowSpec object to **over** on functions such as **rank()**, **dense\_rank()**, **sum()** etc
* Syntax: **rank().over(spec) where spec = Window.partitionBy(‘ColumnName’)**
* Aggregations – **sum, avg, min, max** etc
* Ranking – **rank, dense\_rank, row\_number** etc
* Windowing – **lead, lag** etc

|  |  |
| --- | --- |
|  | orderItems = spark. \ |
|  | read. \ |
|  | json('/Users/itversity/Research/data/retail\_db\_json/order\_items') |
|  |  |
|  | from pyspark.sql.window import \* |
|  | from pyspark.sql.functions import \* |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | # spec = Window.partitionBy('order\_item\_order\_id') |
|  | spec = Window.partitionBy(orderItems.order\_item\_order\_id) |
|  | orderItemsWithRevenue = orderItems. \ |
|  | withColumn('order\_revenue', round(sum(orderItems.order\_item\_subtotal).over(spec), 2)) |
|  |  |
|  | orderItemsWithRevenue.printSchema() |
|  | orderItemsWithRevenue.show() |

[**view raw**](https://gist.github.com/dgadiraju/56a9b3f3628ece7f185935577ebd5814/raw/47a69bd900e211dfeffff0a29e86e84c79d01566/pyspark-dataframe-operations-window-functions-example.py)[**pyspark-dataframe-operations-window-functions-example.py**](https://gist.github.com/dgadiraju/56a9b3f3628ece7f185935577ebd5814#file-pyspark-dataframe-operations-window-functions-example-py) hosted with  by [**GitHub**](https://github.com/)

## Problem Statement

Let us define the problem statement and see the real usage of Analytics or Windowing functions.

* Problem Statement – Get top N Products Per day
* Get daily product revenue code from the previous topic.
* Use ranking functions and get the rank associated based on revenue for each day
* Once we get rank, let us filter for top n products.

|  |  |
| --- | --- |
|  | [dev] |
|  | executionMode = local |
|  | input.base.dir = /Users/itversity/Research/data/retail\_db |
|  | output.base.dir = /Users/itversity/Research/data/bootcamp/pyspark |
|  |  |
|  | [prod] |
|  | executionMode = yarn-client |
|  | input.base.dir = /public/retail\_db |
|  | output.base.dir = /user/training/bootcamp/pyspark |

[**view raw**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58/raw/d7768968719340ccc67ba6ef639330fe42ae097d/pyspark-dataframes-01-application.properties)[**pyspark-dataframes-01-application.properties**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58#file-pyspark-dataframes-01-application-properties) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | import configparser as cp, sys |
|  | from pyspark.sql import SparkSession |
|  |  |
|  | props = cp.RawConfigParser() |
|  | props.read('src/main/resources/application.properties') |
|  | env = sys.argv[1] |
|  |  |
|  | spark = SparkSession.\ |
|  | builder.\ |
|  | appName('Daily Product Revenue using Data Frame Operations').\ |
|  | master(props.get(env, 'executionMode')).\ |
|  | getOrCreate() |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | inputBaseDir = props.get(env, 'input.base.dir') |
|  | orders = spark.read. \ |
|  | format('csv'). \ |
|  | schema('''order\_id INT, |
|  | order\_date STRING, |
|  | order\_customer\_id INT, |
|  | order\_status STRING |
|  | '''). \ |
|  | load(inputBaseDir + '/orders') |
|  |  |
|  | orderItems = spark.read. \ |
|  | format('csv'). \ |
|  | schema('''order\_item\_id INT, |
|  | order\_item\_order\_id INT, |
|  | order\_item\_product\_id INT, |
|  | order\_item\_quantity INT, |
|  | order\_item\_subtotal FLOAT, |
|  | order\_item\_product\_price FLOAT |
|  | '''). \ |
|  | load(inputBaseDir + '/order\_items') |
|  |  |
|  | from pyspark.sql.functions import sum, round |
|  | dailyProductRevenue = orders. \ |
|  | where('order\_status in ("COMPLETE", "CLOSED")'). \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id). \ |
|  | groupBy('order\_date', 'order\_item\_product\_id'). \ |
|  | agg(round(sum(orderItems.order\_item\_subtotal), 2).alias('revenue')) |
|  |  |
|  | dailyProductRevenueSorted = dailyProductRevenue. \ |
|  | orderBy(dailyProductRevenue.order\_date, dailyProductRevenue.revenue.desc()) |
|  |  |
|  | outputBaseDir = props.get(env, 'output.base.dir') |
|  | dailyProductRevenueSorted.write.csv(outputBaseDir + '/daily\_product\_revenue') |

[**view raw**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58/raw/d7768968719340ccc67ba6ef639330fe42ae097d/pyspark-dataframes-02-daily-product-revenue.py)[**pyspark-dataframes-02-daily-product-revenue.py**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58#file-pyspark-dataframes-02-daily-product-revenue-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | spark-submit \ |
|  | --master yarn \ |
|  | --deploy-mode client \ |
|  | --conf spark.ui.port=12901 \ |
|  | src/main/python/retail\_db/df/DailyProductRevenueDFO.py \ |
|  | prod |

[**view raw**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58/raw/d7768968719340ccc67ba6ef639330fe42ae097d/pyspark-dataframes-03-daily-product-revenue.sh)[**pyspark-dataframes-03-daily-product-revenue.sh**](https://gist.github.com/dgadiraju/cd03cdbeb725d72c1b109f282e631f58#file-pyspark-dataframes-03-daily-product-revenue-sh) hosted with  by [**GitHub**](https://github.com/)

## Creating Window Spec

Let us see how to create Window Spec.

* Window have APIs such as partitionBy, orderBy
* For aggregations, we can define the group by using partitionBy
* For ranking or windowing, we need to use partitionBy and then orderBy. partitionBy is to group the data and orderBy is to sort the data to assign rank.
* partitionBy or orderBy returns WindowSpec object
* WindowSpec object needs to be passed to over with ranking and aggregate functions.

## Performing aggregations

Let us see how to perform aggregations where data is partitioned by a key (such as a department).

* We have functions such as sum, avg, min, max etc which can be used to aggregate the data.
* We need to create WindowSpec object using partitionBy to get aggregations within each group.

|  |  |
| --- | --- |
|  | # employeesPath = '/Users/itversity/Research/data/hr\_db/employees/part-00000' |
|  | employeesPath = '/mnt/c/data/hr\_db/employees/part-00000' |
|  |  |
|  | employees = spark. \ |
|  | read. \ |
|  | format('csv'). \ |
|  | option('sep', '\t'). \ |
|  | schema('''employee\_id INT, |
|  | first\_name STRING, |
|  | last\_name STRING, |
|  | email STRING, |
|  | phone\_number STRING, |
|  | hire\_date STRING, |
|  | job\_id STRING, |
|  | salary FLOAT, |
|  | commission\_pct STRING, |
|  | manager\_id STRING, |
|  | department\_id STRING |
|  | '''). \ |
|  | load(employeesPath) |
|  |  |
|  | employees.show() |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |

[**view raw**](https://gist.github.com/dgadiraju/4f756c9f79c7ded5c48871bd1ed5a2f2/raw/5971f065c4b751be38fd4d1329b59452ff296f7e/spark-dataframes-aggregations-01-read-data.py)[**spark-dataframes-aggregations-01-read-data.py**](https://gist.github.com/dgadiraju/4f756c9f79c7ded5c48871bd1ed5a2f2#file-spark-dataframes-aggregations-01-read-data-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | from pyspark.sql.window import \* |
|  |  |
|  | spec = Window.partitionBy('department\_id') |

[**view raw**](https://gist.github.com/dgadiraju/4f756c9f79c7ded5c48871bd1ed5a2f2/raw/5971f065c4b751be38fd4d1329b59452ff296f7e/spark-dataframes-aggregations-02-define-spec.py)[**spark-dataframes-aggregations-02-define-spec.py**](https://gist.github.com/dgadiraju/4f756c9f79c7ded5c48871bd1ed5a2f2#file-spark-dataframes-aggregations-02-define-spec-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ''' |
|  | SELECT employee\_id, salary, department\_id, |
|  | sum(salary) OVER (PARTITION BY department\_id) department\_salary\_expense |
|  | FROM employees |
|  | ORDER BY department\_id, salary DESC; |
|  | ''' |
|  |  |
|  | from pyspark.sql.functions import \* |
|  |  |
|  | employeesSalary = employees. \ |
|  | select('employee\_id', 'salary', 'department\_id'). \ |
|  | withColumn('department\_salary\_expense', sum(col('salary')).over(spec)). \ |
|  | orderBy(col('department\_id'), col('salary').desc()) |
|  |  |
|  | employeesSalary.show() |
|  |  |
|  | # We can directly invoke over with WindowSpec object |
|  | # without creating variable |
|  | employeesSalary = employees. \ |
|  | select('employee\_id', 'salary', 'department\_id'). \ |
|  | withColumn('department\_salary\_expense', sum(col('salary')).over(Window.partitionBy('department\_id'))). \ |
|  | orderBy(col('department\_id'), col('salary').desc()) |
|  |  |
|  | employeesSalary.show() |

[**view raw**](https://gist.github.com/dgadiraju/4f756c9f79c7ded5c48871bd1ed5a2f2/raw/5971f065c4b751be38fd4d1329b59452ff296f7e/spark-dataframes-aggregations-03-using-sum.py)[**spark-dataframes-aggregations-03-using-sum.py**](https://gist.github.com/dgadiraju/4f756c9f79c7ded5c48871bd1ed5a2f2#file-spark-dataframes-aggregations-03-using-sum-py) hosted with  by [**GitHub**](https://github.com/)

* Some realistic use cases
  + Get the average salary for each department and get all employee details who earn more than the average salary
  + Get average revenue for each day and get all the orders who earn revenue more than average revenue
  + Get the highest order revenue and get all the orders which have revenue more than 75% of the revenue

## Using Windowing Functions

Let us see details about windowing functions where data is partitioned by a key (such as a department) and then sorted by some other key (such as hire date).

* We have functions such as lead, lag, first, last etc
* We need to create WindowSpec object using partitionBy and then orderBy for most of the windowing functions
* lead and lag take any column using which you want to get information based on partition and order columns.
* Some realistic use cases
  + The salary difference between current and next/previous employee within each department
  + Time Series – Revenue comparison between two windows

|  |  |
| --- | --- |
|  | # employeesPath = '/Users/itversity/Research/data/hr\_db/employees/part-00000' |
|  | employeesPath = '/mnt/c/data/hr\_db/employees/part-00000' |
|  |  |
|  | employees = spark. \ |
|  | read. \ |
|  | format('csv'). \ |
|  | option('sep', '\t'). \ |
|  | schema('''employee\_id INT, |
|  | first\_name STRING, |
|  | last\_name STRING, |
|  | email STRING, |
|  | phone\_number STRING, |
|  | hire\_date STRING, |
|  | job\_id STRING, |
|  | salary FLOAT, |
|  | commission\_pct STRING, |
|  | manager\_id STRING, |
|  | department\_id STRING |
|  | '''). \ |
|  | load(employeesPath) |
|  |  |
|  | employees.show() |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |

[**view raw**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210/raw/488ad85ab2a7eae63ac57db8e01cbda41b4c9cf3/spark-dataframes-windowing-01-read-data.py)[**spark-dataframes-windowing-01-read-data.py**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210#file-spark-dataframes-windowing-01-read-data-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | from pyspark.sql.window import \* |
|  |  |
|  | spec = Window. \ |
|  | partitionBy('department\_id'). \ |
|  | orderBy(employees.salary.desc()) |

[**view raw**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210/raw/488ad85ab2a7eae63ac57db8e01cbda41b4c9cf3/spark-dataframes-windowing-02-define-spec.py)[**spark-dataframes-windowing-02-define-spec.py**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210#file-spark-dataframes-windowing-02-define-spec-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ''' |
|  | SELECT employee\_id, salary, department\_id, |
|  | lead(salary, 1) OVER (PARTITION BY department\_id ORDER BY salary DESC) lead\_salary |
|  | FROM employees |
|  | ORDER BY department\_id, salary DESC; |
|  | ''' |
|  |  |
|  | from pyspark.sql.functions import lead |
|  |  |
|  | employeesLead = employees. \ |
|  | select('employee\_id', 'salary', 'department\_id'). \ |
|  | withColumn('lead\_salary', lead(employees.salary, 1).over(spec)). \ |
|  | orderBy(employees.department\_id, employees.salary.desc()) |
|  |  |
|  | employeesLead.show(200) |

[**view raw**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210/raw/488ad85ab2a7eae63ac57db8e01cbda41b4c9cf3/spark-dataframes-windowing-03-lead.py)[**spark-dataframes-windowing-03-lead.py**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210#file-spark-dataframes-windowing-03-lead-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ''' |
|  | SELECT employee\_id, salary, department\_id, |
|  | lag(salary) OVER (PARTITION BY department\_id ORDER BY salary DESC) lag\_salary |
|  | FROM employees |
|  | ORDER BY department\_id, salary DESC; |
|  | ''' |
|  |  |
|  | from pyspark.sql.functions import lag |
|  |  |
|  | employeesLag = employees. \ |
|  | select('employee\_id', 'salary', 'department\_id'). \ |
|  | withColumn('lag\_salary', lag(employees.salary, 1).over(spec)). \ |
|  | orderBy(employees.department\_id, employees.salary.desc()) |
|  |  |
|  | employeesLag.show(200) |

[**view raw**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210/raw/488ad85ab2a7eae63ac57db8e01cbda41b4c9cf3/spark-dataframes-windowing-04-lag.py)[**spark-dataframes-windowing-04-lag.py**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210#file-spark-dataframes-windowing-04-lag-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ''' |
|  | SELECT employee\_id, salary, department\_id, |
|  | first\_value(salary) OVER (PARTITION BY department\_id ORDER BY salary DESC) first\_salary |
|  | FROM employees |
|  | ORDER BY department\_id, salary DESC; |
|  | ''' |
|  |  |
|  | from pyspark.sql.functions import first |
|  |  |
|  | employeesFirst = employees. \ |
|  | select('employee\_id', 'salary', 'department\_id'). \ |
|  | withColumn('first\_salary', first(employees.salary).over(spec)). \ |
|  | orderBy(employees.department\_id, employees.salary.desc()) |
|  |  |
|  | employeesFirst.show(200) |
|  |  |
|  |  |

[**view raw**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210/raw/488ad85ab2a7eae63ac57db8e01cbda41b4c9cf3/spark-dataframes-windowing-05-first.py)[**spark-dataframes-windowing-05-first.py**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210#file-spark-dataframes-windowing-05-first-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ''' |
|  | SELECT employee\_id, salary, department\_id, |
|  | last\_value(salary) OVER |
|  | (PARTITION BY department\_id ORDER BY salary DESC |
|  | ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) last\_salary |
|  | FROM employees |
|  | ORDER BY department\_id, salary DESC; |
|  | ''' |
|  |  |
|  | spec = Window. \ |
|  | partitionBy('department\_id'). \ |
|  | orderBy(employees.salary.desc()). \ |
|  | rangeBetween(Window.unboundedPreceding, Window.unboundedFollowing) |
|  |  |
|  | from pyspark.sql.functions import last |
|  |  |
|  | employeesLast = employees. \ |
|  | select('employee\_id', 'salary', 'department\_id'). \ |
|  | withColumn('last\_salary', last(employees.salary, False).over(spec)). \ |
|  | orderBy(employees.department\_id, employees.salary.desc()) |
|  |  |
|  | employeesLast.show(200) |

[**view raw**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210/raw/488ad85ab2a7eae63ac57db8e01cbda41b4c9cf3/spark-dataframes-windowing-06-last.py)[**spark-dataframes-windowing-06-last.py**](https://gist.github.com/dgadiraju/61010493382fec4356dac62c33811210#file-spark-dataframes-windowing-06-last-py) hosted with  by [**GitHub**](https://github.com/)

## Ranking Functions

Let us talk about ranking functions where data is partitioned by a key (such as a department) and then sorted by some other key (such as salary).

* We have functions like rank, dense\_rank, row\_number etc
* We need to create WindowSpec object using partitionBy and then orderBy for most of the ranking functions
* Some realistic use cases
  + Assign rank to employees based on salary within each department
  + Assign ranks to products based on revenue each day or month

|  |  |
| --- | --- |
|  | # employeesPath = '/Users/itversity/Research/data/hr\_db/employees/part-00000' |
|  | employeesPath = '/mnt/c/data/hr\_db/employees/part-00000' |
|  |  |
|  | employees = spark. \ |
|  | read. \ |
|  | format('csv'). \ |
|  | option('sep', '\t'). \ |
|  | schema('''employee\_id INT, |
|  | first\_name STRING, |
|  | last\_name STRING, |
|  | email STRING, |
|  | phone\_number STRING, |
|  | hire\_date STRING, |
|  | job\_id STRING, |
|  | salary FLOAT, |
|  | commission\_pct STRING, |
|  | manager\_id STRING, |
|  | department\_id STRING |
|  | '''). \ |
|  | load(employeesPath) |
|  |  |
|  | employees.show() |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |

[**view raw**](https://gist.github.com/dgadiraju/43b0963fddafee3862ff4bebfdbf224c/raw/a48c3a28eb635306b4becd9cea361b9997889fdb/spark-dataframes-ranking-01-read-data.py)[**spark-dataframes-ranking-01-read-data.py**](https://gist.github.com/dgadiraju/43b0963fddafee3862ff4bebfdbf224c#file-spark-dataframes-ranking-01-read-data-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | from pyspark.sql.window import \* |
|  |  |
|  | spec = Window. \ |
|  | partitionBy('department\_id'). \ |
|  | orderBy(employees.salary.desc()) |

[**view raw**](https://gist.github.com/dgadiraju/43b0963fddafee3862ff4bebfdbf224c/raw/a48c3a28eb635306b4becd9cea361b9997889fdb/spark-dataframes-ranking-02-define-spec.py)[**spark-dataframes-ranking-02-define-spec.py**](https://gist.github.com/dgadiraju/43b0963fddafee3862ff4bebfdbf224c#file-spark-dataframes-ranking-02-define-spec-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ''' |
|  | SELECT employee\_id, salary, department\_id, |
|  | rank() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank |
|  | FROM employees |
|  | ORDER BY department\_id, salary DESC; |
|  | ''' |
|  |  |
|  | from pyspark.sql.functions import rank |
|  |  |
|  | employeesRanked = employees. \ |
|  | select('employee\_id', 'salary', 'department\_id'). \ |
|  | withColumn('rank', rank().over(spec)). \ |
|  | orderBy(employees.department\_id, employees.salary.desc()) |
|  |  |
|  | employeesRanked.show(200) |

[**view raw**](https://gist.github.com/dgadiraju/43b0963fddafee3862ff4bebfdbf224c/raw/a48c3a28eb635306b4becd9cea361b9997889fdb/spark-dataframes-ranking-03-rank.py)[**spark-dataframes-ranking-03-rank.py**](https://gist.github.com/dgadiraju/43b0963fddafee3862ff4bebfdbf224c#file-spark-dataframes-ranking-03-rank-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ''' |
|  | SELECT employee\_id, salary, department\_id, |
|  | dense\_rank() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank |
|  | FROM employees |
|  | ORDER BY department\_id, salary DESC; |
|  | ''' |
|  |  |
|  | from pyspark.sql.functions import dense\_rank |
|  |  |
|  | employeesDenseRanked = employees. \ |
|  | select('employee\_id', 'salary', 'department\_id'). \ |
|  | withColumn('rank', dense\_rank().over(spec)). \ |
|  | orderBy(employees.department\_id, employees.salary.desc()) |
|  |  |
|  | employeesDenseRanked.show(200) |

[**view raw**](https://gist.github.com/dgadiraju/43b0963fddafee3862ff4bebfdbf224c/raw/a48c3a28eb635306b4becd9cea361b9997889fdb/spark-dataframes-ranking-04-dense-rank.py)[**spark-dataframes-ranking-04-dense-rank.py**](https://gist.github.com/dgadiraju/43b0963fddafee3862ff4bebfdbf224c#file-spark-dataframes-ranking-04-dense-rank-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ''' |
|  | SELECT employee\_id, salary, department\_id, |
|  | row\_number() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rn |
|  | FROM employees |
|  | ORDER BY department\_id, salary DESC; |
|  | ''' |
|  |  |
|  | from pyspark.sql.functions import row\_number |
|  |  |
|  | employeesRowNumbered = employees. \ |
|  | select('employee\_id', 'salary', 'department\_id'). \ |
|  | withColumn('rn', row\_number().over(spec)). \ |
|  | orderBy(employees.department\_id, employees.salary.desc()) |
|  |  |
|  | employeesRowNumbered.show(200) |

[**view raw**](https://gist.github.com/dgadiraju/43b0963fddafee3862ff4bebfdbf224c/raw/a48c3a28eb635306b4becd9cea361b9997889fdb/spark-dataframes-ranking-05-rownumber.py)[**spark-dataframes-ranking-05-rownumber.py**](https://gist.github.com/dgadiraju/43b0963fddafee3862ff4bebfdbf224c#file-spark-dataframes-ranking-05-rownumber-py) hosted with  by [**GitHub**](https://github.com/)

## Development Life Cycle

Let us discuss details related to the development life cycle for our problem statement.

* Problem Statement – Get top N Products Per day
* Take the DailyProductRevenue code which gives us order\_date, order\_item\_product\_id, and revenue
* Import Window and create a spec to partition by date and order by revenue in descending order.
* Use withColumn and assign the rank
* Filter data where rank is less than or equal to topN passed as an argument to the program
* Drop rank field as we do not want to save the data and then sort in ascending order by date and descending order by revenue
* Save the data frame into a file

|  |  |
| --- | --- |
|  | [dev] |
|  | executionMode = local |
|  | input.base.dir = /Users/itversity/Research/data/retail\_db |
|  | output.base.dir = /Users/itversity/Research/data/bootcamp/pyspark |
|  |  |
|  | [prod] |
|  | executionMode = yarn-client |
|  | input.base.dir = /public/retail\_db |
|  | output.base.dir = /user/training/bootcamp/pyspark |

[**view raw**](https://gist.github.com/dgadiraju/322adf77162acf436b7600bd5878499e/raw/b0027b233be80cfca0ded8463d7d115561fb7241/pyspark-dataframes-01-application.properties)[**pyspark-dataframes-01-application.properties**](https://gist.github.com/dgadiraju/322adf77162acf436b7600bd5878499e#file-pyspark-dataframes-01-application-properties) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | import configparser as cp, sys |
|  |  |
|  | from pyspark.sql import SparkSession |
|  | from pyspark.sql.functions import sum, round, dense\_rank |
|  | from pyspark.sql.window import \* |
|  |  |
|  | props = cp.RawConfigParser() |
|  | props.read('src/main/resources/application.properties') |
|  | env = sys.argv[1] |
|  | topN = int(sys.argv[2]) |
|  |  |
|  | spark = SparkSession. \ |
|  | builder. \ |
|  | master(props.get(env, 'executionMode')). \ |
|  | appName('Get TopN Daily Products using Data Frame Operations'). \ |
|  | getOrCreate() |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  | spark.sparkContext.setLogLevel('ERROR') |
|  |  |
|  | inputBaseDir = props.get(env, 'input.base.dir') |
|  | orders = spark.read. \ |
|  | format('csv'). \ |
|  | schema('order\_id int, order\_date string, order\_customer\_id int, order\_status string'). \ |
|  | load(inputBaseDir + '/orders') |
|  |  |
|  | orderItems = spark.read. \ |
|  | format('csv'). \ |
|  | schema('''order\_item\_id int, |
|  | order\_item\_order\_id int, |
|  | order\_item\_product\_id int, |
|  | order\_item\_quantity int, |
|  | order\_item\_subtotal float, |
|  | order\_item\_product\_price float |
|  | '''). \ |
|  | load(inputBaseDir + '/order\_items') |
|  |  |
|  | dailyProductRevenue = orders. \ |
|  | where('order\_status in ("COMPLETE", "CLOSED")'). \ |
|  | join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id). \ |
|  | groupBy('order\_date', 'order\_item\_product\_id'). \ |
|  | agg(round(sum('order\_item\_subtotal'), 2).alias('revenue')) |
|  |  |
|  | spec = Window. \ |
|  | partitionBy('order\_date'). \ |
|  | orderBy(dailyProductRevenue.revenue.desc()) |
|  |  |
|  | dailyProductRevenueRanked = dailyProductRevenue. \ |
|  | withColumn("rnk", dense\_rank().over(spec)) |
|  |  |
|  | topNDailyProducts = dailyProductRevenueRanked. \ |
|  | where(dailyProductRevenueRanked.rnk <= topN). \ |
|  | drop('rnk'). \ |
|  | orderBy('order\_date', dailyProductRevenueRanked.revenue.desc()) |
|  |  |
|  | outputBaseDir = props.get(env, 'output.base.dir') |
|  | topNDailyProducts. \ |
|  | write. \ |
|  | csv(outputBaseDir + '/topn\_daily\_products') |

[**view raw**](https://gist.github.com/dgadiraju/322adf77162acf436b7600bd5878499e/raw/b0027b233be80cfca0ded8463d7d115561fb7241/pyspark-dataframes-02-top-n-daily-products.py)[**pyspark-dataframes-02-top-n-daily-products.py**](https://gist.github.com/dgadiraju/322adf77162acf436b7600bd5878499e#file-pyspark-dataframes-02-top-n-daily-products-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | spark-submit \ |
|  | --master yarn \ |
|  | --deploy-mode client \ |
|  | --conf spark.ui.port=12901 \ |
|  | src/main/python/retail\_db/df/TopNDailyProductsDFO.py \ |
|  | prod |

[**view raw**](https://gist.github.com/dgadiraju/322adf77162acf436b7600bd5878499e/raw/b0027b233be80cfca0ded8463d7d115561fb7241/pyspark-dataframes-03-top-n-daily-products.sh)[**pyspark-dataframes-03-top-n-daily-products.sh**](https://gist.github.com/dgadiraju/322adf77162acf436b7600bd5878499e#file-pyspark-dataframes-03-top-n-daily-products-sh) hosted with  by [**GitHub**](https://github.com/)

# Spark SQL – Basic Transformations such as filtering, aggregations, joins etc

posted on FEBRUARY 3, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

As part of this session we will see basic transformations we can perform on top of Data Frames such as filtering, aggregations, joins etc using SQL. We will build end to end application by taking a simple problem statement.

* Spark SQL – Overview
* Problem Statement – Get daily product revenue
* Relationship with Hive
* Projecting data using select
* Filtering data using where
* Joining Data Sets
* Grouping data and performing aggregations
* Sorting data

## Spark SQL – Overview

<https://youtu.be/guYnHu9DWkY>

Let us recap about Data Frame Operations. It is one of the 2 ways we can process Data Frames.

* Selection or Projection – select clause
* Filtering data – where clause
* Joins – join (supports outer join as well)
* Aggregations – group by and aggregations with support of functions such as sum, avg, min, max etc
* Sorting – order by
* Analytics Functions – aggregations, ranking and windowing functions

## Problem Statement – Get daily product revenue

Here is the problem statement for which we will be exploring Data Frame APIs to come up with final solution.

* Get daily product revenue
* orders – order\_id, order\_date, order\_customer\_id, order\_status
* order\_items – order\_item\_id, order\_item\_order\_id, order\_item\_product\_id, order\_item\_quantity, order\_item\_subtotal, order\_item\_product\_price
* Data is comma separated
* We will fetch data using spark.read.csv
* Apply type cast functions to convert fields into their original type where ever is applicable.

|  |  |
| --- | --- |
|  | spark = SparkSession. \ |
|  | builder. \ |
|  | master('local'). \ |
|  | appName('CSV Example'). \ |
|  | getOrCreate() |
|  |  |
|  | orders = spark.read. \ |
|  | format('csv'). \ |
|  | schema('order\_id int, order\_date string, order\_customer\_id int, order\_status string'). \ |
|  | load('/Users/itversity/Research/data/retail\_db/orders') |
|  |  |
|  | orders.printSchema() |
|  | orders.show() |
|  |  |
|  | orderItems = spark.read. \ |
|  | format('csv'). \ |
|  | schema('''order\_item\_id int, |
|  | order\_item\_order\_id int, |
|  | order\_item\_product\_id int, |
|  | order\_item\_quantity int, |
|  | order\_item\_subtotal float, |
|  | order\_item\_product\_price float |
|  | '''). \ |
|  | load('/Users/itversity/Research/data/retail\_db/order\_items') |
|  |  |
|  | orderItems.printSchema() |
|  | orderItems.show() |

[**view raw**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7/raw/af963c48c5cb1a613ff7cb6fab3bd16b97fb1999/pyspark-dataframe-01-csv-example.py)[**pyspark-dataframe-01-csv-example.py**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7#file-pyspark-dataframe-01-csv-example-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # In case you are using pycharm, first you need to create object of type SparkSession |
|  | spark = SparkSession. \ |
|  | builder. \ |
|  | master('local'). \ |
|  | appName('CSV Example'). \ |
|  | getOrCreate() |
|  |  |
|  | ordersCSV = spark.read. \ |
|  | csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | orderItemsCSV = spark.read. \ |
|  | csv('/public/retail\_db/order\_items'). \ |
|  | toDF('order\_item\_id', 'order\_item\_order\_id', 'order\_item\_product\_id', |
|  | 'order\_item\_quantity', 'order\_item\_subtotal', 'order\_item\_product\_price') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  |  |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.printSchema() |
|  | orders.show() |
|  |  |
|  | orderItems = orderItemsCSV.\ |
|  | withColumn('order\_item\_id', orderItemsCSV.order\_item\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_order\_id', orderItemsCSV.order\_item\_order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_product\_id', orderItemsCSV.order\_item\_product\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_quantity', orderItemsCSV.order\_item\_quantity.cast(IntegerType())). \ |
|  | withColumn('order\_item\_subtotal', orderItemsCSV.order\_item\_subtotal.cast(FloatType())). \ |
|  | withColumn('order\_item\_product\_price', orderItemsCSV.order\_item\_product\_price.cast(FloatType())) |
|  |  |
|  | orderItems.printSchema() |
|  | orderItems.show() |

[**view raw**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7/raw/af963c48c5cb1a613ff7cb6fab3bd16b97fb1999/pyspark-dataframe-02-csv-example.py)[**pyspark-dataframe-02-csv-example.py**](https://gist.github.com/dgadiraju/22f7907bc0e8d014c138f93ee23ef3d7#file-pyspark-dataframe-02-csv-example-py) hosted with  by [**GitHub**](https://github.com/)

* We can register both orders and orderItems as temporary views.
  + Switch to database in hive – spark.sql('use bootcampdemo')
  + orders as orders – orders.createOrReplaceTempView('orders')
  + orderItems as order\_items – orderItems.createOrReplaceTempView('order\_items')
  + List tables – spark.sql('show tables').show()
  + Describe table – spark.sql('describe orders').show()

## Relationship with Hive

Let us see how Spark is related to Hive.

* Hive is a logical database on top of HDFS
* All hive databases, tables and even partitions are nothing but directories in HDFS
* We can create tables in Hive with column names and data types
* Table names, column names, data types, location, file format, delimiter information is considered as metadata
* This metadata is stored in metastore which is typically relational database such as MySQL, Postgres, Oracle etc
* Once table is created, data can be queried or processed using HiveQL
* HiveQL will be compiled into Spark or Map Reduce job based on the execution engine.
* If Hive is integrated with Spark on the cluster using SparkSession object’s sql API we should be able to query and process data from Hive tables using Spark engine
* Query output will be converted to Data Frame
* SparkSession object’s sql API can execute standard hive commands such as show tables, show functions etc
* Standard Hive commands (except SQL)
  + spark is of type SparkSession
  + List of tables – spark.sql("show tables").show()
  + Switch database – spark.sql("use bootcampdemo").show()
  + Describe table – spark.sql("describe table orders").show()
  + Show functions – for f in spark.sql("show functions").collect(): print(f)
  + Describe function – for f in spark.sql("describe function substring").collect(): print(f)
* We can also create/drop tables, insert/load data into tables using Hive syntax as part of sql function of SparkSession object
* As part of SparkSession object’s read, there is an API which facilitate us to read raw data from Hive table into Data Frame
* write package of data frame provides us APIs such as saveAsTable, insertInto etc to directly write data frame into Hive table.

## Selection or Projection – select clause

Now let us see how we can project data the way we want using select.

* We can run queries directly from hive tables or register data frames as temporary views/tables.
* We can use select and fetch data from the fields we are looking for.
* We can represent data using DataFrame.ColumnName or directly ‘ColumnName’ in select clause – e.g.: spark.sql('select order\_id, order\_date from orders').show()
* We can apply necessary functions to manipulate data while it is being projected – spark.sql('select substring(order\_date, 1, 7) from orders').show()
* We can give aliases to the derived fields using alias function – spark.sql('select substring(order\_date, 1, 7) as order\_month from orders').show()

## Filtering data – where clause

We can use where clause to filter the data.

* One by using class.attributeName and comparing with values – e. g.: spark.sql('select \* from orders where order\_status = "COMPLETE"').show()
* Make sure both orders and orderItems data frames are created
* Let us see few more examples
  + Get orders which are either COMPLETE or CLOSED
  + Get orders which are either COMPLETE or CLOSED and placed in month of 2013 August
  + Get order items where order\_item\_subtotal is not equal to product of order\_item\_quantity and order\_item\_product\_price
  + Get all the orders which are placed on first of every month

|  |  |
| --- | --- |
|  | # Get orders which are either COMPLETE or CLOSED |
|  | spark.sql('select \* from orders where order\_status = "COMPLETE" or order\_status = "CLOSED"').show() |
|  | spark.sql('select \* from orders where order\_status in ("COMPLETE", "CLOSED")').show() |

[**view raw**](https://gist.github.com/dgadiraju/4b020beb9c73ca30fd5f51fdf02cc026/raw/70e3fbeed6cb69478c026a0081523388b4fd4e29/pyspark-sparksql-filtering-01.py)[**pyspark-sparksql-filtering-01.py**](https://gist.github.com/dgadiraju/4b020beb9c73ca30fd5f51fdf02cc026#file-pyspark-sparksql-filtering-01-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get orders which are either COMPLETE or CLOSED and placed in month of 2013 August |
|  |  |
|  | spark.sql('select \* from orders where order\_status in ("COMPLETE", "CLOSED") and order\_date like "2013-08%"').show() |

[**view raw**](https://gist.github.com/dgadiraju/4b020beb9c73ca30fd5f51fdf02cc026/raw/70e3fbeed6cb69478c026a0081523388b4fd4e29/pyspark-sparksql-filtering-02.py)[**pyspark-sparksql-filtering-02.py**](https://gist.github.com/dgadiraju/4b020beb9c73ca30fd5f51fdf02cc026#file-pyspark-sparksql-filtering-02-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get order items where order\_item\_subtotal is not equal to product of order\_item\_quantity and order\_item\_product\_price |
|  | spark.sql('''select \* from order\_items where |
|  | order\_item\_subtotal != round(order\_item\_quantity \* order\_item\_product\_price, 2)''').show() |

[**view raw**](https://gist.github.com/dgadiraju/4b020beb9c73ca30fd5f51fdf02cc026/raw/70e3fbeed6cb69478c026a0081523388b4fd4e29/pyspark-sparksql-filtering-03.py)[**pyspark-sparksql-filtering-03.py**](https://gist.github.com/dgadiraju/4b020beb9c73ca30fd5f51fdf02cc026#file-pyspark-sparksql-filtering-03-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get all the orders which are placed on first of every month |
|  | spark.sql('''select \* from orders |
|  | where date\_format(order\_date, "dd") = "01"''').show() |

[**view raw**](https://gist.github.com/dgadiraju/4b020beb9c73ca30fd5f51fdf02cc026/raw/70e3fbeed6cb69478c026a0081523388b4fd4e29/pyspark-sparksql-filtering-04.py)[**pyspark-sparksql-filtering-04.py**](https://gist.github.com/dgadiraju/4b020beb9c73ca30fd5f51fdf02cc026#file-pyspark-sparksql-filtering-04-py) hosted with  by [**GitHub**](https://github.com/)

## Joining Data Sets

Quite often we need to deal with multiple data sets which are related with each other.

* We need to first understand the relationship with respect to data sets
* All our data sets have relationships defined between them.
  + orders and order\_items are transaction tables. orders is parent and order\_items is child. Relationship is established between the two using order\_id (in order\_items, it is represented as order\_item\_order\_id)
  + We also have product catalog normalized into 3 tables – products, categories and departments (with relationships established in that order)
  + We also have customers table
  + There is relationship between customers and orders – customers is parent data set as one customer can place multiple orders.
  + There is relationship between product catalog and order\_items via products – products is parent data set as one product can be ordered as part of multiple order\_items.
* Determine the type of join – inner or outer (left or right or full)
* We can perform joins using ascii syntax with join along with on clause
* We can also perform outer joins (left or right or full)
* Let us see few examples
  + Get all the order items corresponding to COMPLETE or CLOSED orders
  + Get all the orders where there are no corresponding order\_items
  + Check if there are any order\_items where there is no corresponding order in orders data set

|  |  |
| --- | --- |
|  | # Get all the order items corresponding to COMPLETE or CLOSED orders |
|  |  |
|  | spark.sql('select \* from orders o join order\_items oi ' |
|  | 'on o.order\_id = oi.order\_item\_order\_id ' |
|  | 'where o.order\_status in ("COMPLETE", "CLOSED")'). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/ccf75c799467c1e346d8917860d50019/raw/5b195440bb119a14e06625b57ada8b4727b0f9bb/pyspark-sparksql-join-01.py)[**pyspark-sparksql-join-01.py**](https://gist.github.com/dgadiraju/ccf75c799467c1e346d8917860d50019#file-pyspark-sparksql-join-01-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get all the orders where there are no corresponding order\_items |
|  |  |
|  | spark.sql('select \* from orders o left outer join order\_items oi ' |
|  | 'on o.order\_id = oi.order\_item\_order\_id ' |
|  | 'where oi.order\_item\_order\_id is null'). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/ccf75c799467c1e346d8917860d50019/raw/5b195440bb119a14e06625b57ada8b4727b0f9bb/pyspark-sparksql-join-02.py)[**pyspark-sparksql-join-02.py**](https://gist.github.com/dgadiraju/ccf75c799467c1e346d8917860d50019#file-pyspark-sparksql-join-02-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Check if there are any order\_items where there is no corresponding order in orders data set |
|  |  |
|  | spark.sql('select \* from orders o right outer join order\_items oi ' |
|  | 'on o.order\_id = oi.order\_item\_order\_id ' |
|  | 'where o.order\_id is null'). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/ccf75c799467c1e346d8917860d50019/raw/5b195440bb119a14e06625b57ada8b4727b0f9bb/pyspark-sparksql-join-03.py)[**pyspark-sparksql-join-03.py**](https://gist.github.com/dgadiraju/ccf75c799467c1e346d8917860d50019#file-pyspark-sparksql-join-03-py) hosted with  by [**GitHub**](https://github.com/)

## Aggregations using group by and functions

Many times we want to perform aggregations such as sum, average, minimum, maximum etc with in each group. We need to first group the data and then perform aggregation.

* group by is the function which can be used to group the data on one or more columns
* Once data is grouped we can perform all supported aggregations – sum, avg, min, max etc
* Let us see few examples
  + Get count by status from orders
  + Get revenue for each order id from order items
  + Get daily product revenue (order\_date and order\_item\_product\_id are part of keys, order\_item\_subtotal is used for aggregation)

|  |  |
| --- | --- |
|  | # Get count by status from orders |
|  | spark.sql('select order\_status, count(1) status\_count ' |
|  | 'from orders group by order\_status'). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/c880c4a23adc8cdd18e66e8ba3cbd265/raw/d6dc8026c02725cbbbe3da0d3deb2787784d64cd/pyspark-sparksql-group-and-agg-01.py)[**pyspark-sparksql-group-and-agg-01.py**](https://gist.github.com/dgadiraju/c880c4a23adc8cdd18e66e8ba3cbd265#file-pyspark-sparksql-group-and-agg-01-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get revenue for each order id from order items |
|  | spark.sql('select order\_item\_order\_id, sum(order\_item\_subtotal) order\_revenue ' |
|  | 'from order\_items group by order\_item\_order\_id'). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/c880c4a23adc8cdd18e66e8ba3cbd265/raw/d6dc8026c02725cbbbe3da0d3deb2787784d64cd/pyspark-sparksql-group-and-agg-02.py)[**pyspark-sparksql-group-and-agg-02.py**](https://gist.github.com/dgadiraju/c880c4a23adc8cdd18e66e8ba3cbd265#file-pyspark-sparksql-group-and-agg-02-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Get daily product revenue |
|  | # filter for complete and closed orders |
|  | # groupBy order\_date and order\_item\_product\_id |
|  | # Use agg and sum on order\_item\_subtotal to get revenue |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | spark.sql('select o.order\_date, oi.order\_item\_product\_id, ' |
|  | 'sum(oi.order\_item\_subtotal) order\_revenue ' |
|  | 'from orders o join order\_items oi ' |
|  | 'on o.order\_id = oi.order\_item\_order\_id ' |
|  | 'where o.order\_status in ("COMPLETE", "CLOSED") ' |
|  | 'group by o.order\_date, oi.order\_item\_product\_id'). \ |
|  | show() |

[**view raw**](https://gist.github.com/dgadiraju/c880c4a23adc8cdd18e66e8ba3cbd265/raw/d6dc8026c02725cbbbe3da0d3deb2787784d64cd/pyspark-sparksql-group-and-agg-03.py)[**pyspark-sparksql-group-and-agg-03.py**](https://gist.github.com/dgadiraju/c880c4a23adc8cdd18e66e8ba3cbd265#file-pyspark-sparksql-group-and-agg-03-py) hosted with  by [**GitHub**](https://github.com/)

## Sorting data

Now let us see how we can sort the data using sort or orderBy.

* order by can be used to sort the data
* We can perform composite sorting by using multiple fields
* By default data will be sorted in ascending order
* We can change the order by using desc
* Let us see few examples
  + Sort orders by status
  + Sort orders by date and then by status
  + Sort order items by order\_item\_order\_id and order\_item\_subtotal descending
  + Take daily product revenue data and sort in ascending order by date and then descending order by revenue.

|  |  |
| --- | --- |
|  | # Sort orders by status |
|  | spark.sql('''select \* from orders |
|  | order by order\_status''').show() |

[**view raw**](https://gist.github.com/dgadiraju/805cf168829c3b1183db4eb992d32404/raw/404ce9c4b4ba5d2ad1b9aeaf916e82729ed75990/pyspark-sparksql-sorting-01.py)[**pyspark-sparksql-sorting-01.py**](https://gist.github.com/dgadiraju/805cf168829c3b1183db4eb992d32404#file-pyspark-sparksql-sorting-01-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | #Sort orders by date and then by status |
|  | spark.sql('''select \* from orders |
|  | order by order\_date, order\_status''').show() |

[**view raw**](https://gist.github.com/dgadiraju/805cf168829c3b1183db4eb992d32404/raw/404ce9c4b4ba5d2ad1b9aeaf916e82729ed75990/pyspark-sparksql-sorting-02.py)[**pyspark-sparksql-sorting-02.py**](https://gist.github.com/dgadiraju/805cf168829c3b1183db4eb992d32404#file-pyspark-sparksql-sorting-02-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Sort order items by order\_item\_order\_id and order\_item\_subtotal descending |
|  | spark.sql('''select \* from order\_items |
|  | order by order\_item\_order\_id, order\_item\_subtotal desc''').show() |

[**view raw**](https://gist.github.com/dgadiraju/805cf168829c3b1183db4eb992d32404/raw/404ce9c4b4ba5d2ad1b9aeaf916e82729ed75990/pyspark-sparksql-sorting-03.py)[**pyspark-sparksql-sorting-03.py**](https://gist.github.com/dgadiraju/805cf168829c3b1183db4eb992d32404#file-pyspark-sparksql-sorting-03-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Take daily product revenue data and |
|  | # sort in ascending order by date and |
|  | # then descending order by revenue. |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | dailyProductRevenue = spark.sql('''select o.order\_date, oi.order\_item\_product\_id, |
|  | round(sum(oi.order\_item\_subtotal), 2) as revenue |
|  | from orders o join order\_items oi |
|  | on o.order\_id = oi.order\_item\_order\_id |
|  | where o.order\_status in ("COMPLETE", "CLOSED") |
|  | group by o.order\_date, oi.order\_item\_product\_id |
|  | order by o.order\_date, revenue desc''') |
|  |  |
|  | dailyProductRevenue.show() |

[**view raw**](https://gist.github.com/dgadiraju/805cf168829c3b1183db4eb992d32404/raw/404ce9c4b4ba5d2ad1b9aeaf916e82729ed75990/pyspark-sparksql-sorting-04.py)[**pyspark-sparksql-sorting-04.py**](https://gist.github.com/dgadiraju/805cf168829c3b1183db4eb992d32404#file-pyspark-sparksql-sorting-04-py) hosted with  by [**GitHub**](https://github.com/)

# Spark SQL – Analytics Functions or Windowing Functions

posted on FEBRUARY 3, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

As part of this session we will see advanced operations such as aggregations, ranking and windowing functions with in each group using clauses such as over, partition by, order by etc. We will also build a solution for problem and run it on multinode cluster.

* Aggregations, Ranking and Windowing Functions
* Problem Statement – Get top n products per day
* Understanding over, partition by and order by clauses
* Performing aggregations
* Using windowing functions
* Ranking with in each partition or group
* Development Life Cycle

## Development Life Cycle (daily product revenue)

<https://youtu.be/UErcF4xupiQ>

Let us develop the application using Pycharm and run it on the cluster.

* Make sure application.properties have required input path and output path along with execution mode
* Read orders and order\_items data into data frames
* Filter for complete and closed orders
* Join with order\_items
* Aggregate to get revenue for each order\_date and order\_item\_product\_id
* Sort in ascending order by date and then descending order by revenue
* Save the output as CSV format
* Validate using Pycharm
* Ship it to the cluster, run it on the cluster and validate.

|  |  |
| --- | --- |
|  | [dev] |
|  | executionMode = local |
|  | input.base.dir = /Users/itversity/Research/data/retail\_db |
|  | output.base.dir = /Users/itversity/Research/data/bootcamp/pyspark |
|  |  |
|  | [prod] |
|  | executionMode = yarn-client |
|  | input.base.dir = /public/retail\_db |
|  | output.base.dir = /user/training/bootcamp/pyspark |

[**view raw**](https://gist.github.com/dgadiraju/4541d492873f4e36a5a0fe54e48b54a1/raw/5d371b41b6a85994f68408370c0aa5412958dca9/pyspark-sparksql-01-application.properties)[**pyspark-sparksql-01-application.properties**](https://gist.github.com/dgadiraju/4541d492873f4e36a5a0fe54e48b54a1#file-pyspark-sparksql-01-application-properties) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | import configparser as cp, sys |
|  | from pyspark.sql import SparkSession |
|  |  |
|  | props = cp.RawConfigParser() |
|  | props.read('src/main/resources/application.properties') |
|  | env = sys.argv[1] |
|  |  |
|  | spark = SparkSession.\ |
|  | builder.\ |
|  | appName("Daily Product Revenue using Data Frames and Spark SQL").\ |
|  | master(props.get(env, 'executionMode')).\ |
|  | getOrCreate() |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | inputBaseDir = props.get(env, 'input.base.dir') |
|  | ordersCSV = spark.read. \ |
|  | csv(inputBaseDir + '/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | orderItemsCSV = spark.read. \ |
|  | csv(inputBaseDir + '/order\_items'). \ |
|  | toDF('order\_item\_id', 'order\_item\_order\_id', 'order\_item\_product\_id', |
|  | 'order\_item\_quantity', 'order\_item\_subtotal', 'order\_item\_product\_price') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  |  |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orderItems = orderItemsCSV.\ |
|  | withColumn('order\_item\_id', orderItemsCSV.order\_item\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_order\_id', orderItemsCSV.order\_item\_order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_product\_id', orderItemsCSV.order\_item\_product\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_quantity', orderItemsCSV.order\_item\_quantity.cast(IntegerType())). \ |
|  | withColumn('order\_item\_subtotal', orderItemsCSV.order\_item\_subtotal.cast(FloatType())). \ |
|  | withColumn('order\_item\_product\_price', orderItemsCSV.order\_item\_product\_price.cast(FloatType())) |
|  |  |
|  | orders.createTempView('orders') |
|  | orderItems.createTempView('order\_items') |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | dailyProductRevenue = spark.sql('''select o.order\_date, oi.order\_item\_product\_id, |
|  | round(sum(oi.order\_item\_subtotal), 2) as revenue |
|  | from orders o join order\_items oi |
|  | on o.order\_id = oi.order\_item\_order\_id |
|  | where o.order\_status in ("COMPLETE", "CLOSED") |
|  | group by o.order\_date, oi.order\_item\_product\_id |
|  | order by o.order\_date, revenue desc''') |
|  |  |
|  | outputBaseDir = props.get(env, 'output.base.dir') |
|  | dailyProductRevenue.write.csv(outputBaseDir + '/daily\_product\_revenue\_sql') |

[**view raw**](https://gist.github.com/dgadiraju/4541d492873f4e36a5a0fe54e48b54a1/raw/5d371b41b6a85994f68408370c0aa5412958dca9/pyspark-sparksql-02-daily-product-revenue.py)[**pyspark-sparksql-02-daily-product-revenue.py**](https://gist.github.com/dgadiraju/4541d492873f4e36a5a0fe54e48b54a1#file-pyspark-sparksql-02-daily-product-revenue-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | spark-submit --master yarn \ |
|  | --deploy-mode client \ |
|  | --conf spark.ui.port=12901 \ |
|  | src/main/python/retail\_db/df/DailyProductRevenueDFS.py \ |
|  | prod |

[**view raw**](https://gist.github.com/dgadiraju/4541d492873f4e36a5a0fe54e48b54a1/raw/5d371b41b6a85994f68408370c0aa5412958dca9/pyspark-sparksql-03-daily-product-revenue.sh)[**pyspark-sparksql-03-daily-product-revenue.sh**](https://gist.github.com/dgadiraju/4541d492873f4e36a5a0fe54e48b54a1#file-pyspark-sparksql-03-daily-product-revenue-sh) hosted with  by [**GitHub**](https://github.com/)

## Aggregations, Ranking and Windowing Functions

Let us understand APIs related to aggregations, ranking and windowing functions.

* There are multiple clauses with in SQL to accomplish these
  + over
  + partition by
  + order by
* All aggregate functions, rank functions and windowing functions can be used with over clause to get aggregations per partition or group
* It is mandatory to specify over clause
* e.g.: rank() over(spec) where spec can be partition by or order by or both
* Aggregations – sum, avg, min, max etc
* Ranking – rank, dense\_rank, row\_number etc
* Windowing – lead, lag etc
* We typically use partition by clause for aggregations and then partition by as well as order by for ranking and windowing functions.

|  |  |
| --- | --- |
|  | orderItemsCSV = spark.read. \ |
|  | csv('/public/retail\_db/order\_items'). \ |
|  | toDF('order\_item\_id', 'order\_item\_order\_id', 'order\_item\_product\_id', |
|  | 'order\_item\_quantity', 'order\_item\_subtotal', 'order\_item\_product\_price') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  |  |
|  | orderItems = orderItemsCSV.\ |
|  | withColumn('order\_item\_id', orderItemsCSV.order\_item\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_order\_id', orderItemsCSV.order\_item\_order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_product\_id', orderItemsCSV.order\_item\_product\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_quantity', orderItemsCSV.order\_item\_quantity.cast(IntegerType())). \ |
|  | withColumn('order\_item\_subtotal', orderItemsCSV.order\_item\_subtotal.cast(FloatType())). \ |
|  | withColumn('order\_item\_product\_price', orderItemsCSV.order\_item\_product\_price.cast(FloatType())) |
|  |  |
|  | orderItems.createTempView('order\_items') |

[**view raw**](https://gist.github.com/dgadiraju/7406df149c6de20c81e413af4b2e4288/raw/a9f2c7d66f937cc73e98be2ba5c8258240571744/pyspark-sparksql-01-read-data.py)[**pyspark-sparksql-01-read-data.py**](https://gist.github.com/dgadiraju/7406df149c6de20c81e413af4b2e4288#file-pyspark-sparksql-01-read-data-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | spark.sql('''select oi.order\_item\_id, oi.order\_item\_order\_id, oi.order\_item\_subtotal, |
|  | round(sum(oi.order\_item\_subtotal) over (partition by oi.order\_item\_order\_id), 2) order\_revenue |
|  | from order\_items oi |
|  | ''').show() |

[**view raw**](https://gist.github.com/dgadiraju/7406df149c6de20c81e413af4b2e4288/raw/a9f2c7d66f937cc73e98be2ba5c8258240571744/pyspark-sparksql-02-aggregations.py)[**pyspark-sparksql-02-aggregations.py**](https://gist.github.com/dgadiraju/7406df149c6de20c81e413af4b2e4288#file-pyspark-sparksql-02-aggregations-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | spark.sql('''select oi.order\_item\_id, oi.order\_item\_order\_id, oi.order\_item\_subtotal, |
|  | rank() over |
|  | (partition by oi.order\_item\_order\_id |
|  | order by oi.order\_item\_subtotal desc |
|  | ) rnk |
|  | from order\_items oi |
|  | ''').show() |

[**view raw**](https://gist.github.com/dgadiraju/7406df149c6de20c81e413af4b2e4288/raw/a9f2c7d66f937cc73e98be2ba5c8258240571744/pyspark-sparksql-03-ranking.py)[**pyspark-sparksql-03-ranking.py**](https://gist.github.com/dgadiraju/7406df149c6de20c81e413af4b2e4288#file-pyspark-sparksql-03-ranking-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | spark.sql('''select oi.order\_item\_id, oi.order\_item\_order\_id, oi.order\_item\_subtotal, |
|  | lead(oi.order\_item\_subtotal) |
|  | over (partition by oi.order\_item\_order\_id |
|  | order by oi.order\_item\_subtotal desc |
|  | ) next\_order\_item\_subtotal |
|  | from order\_items oi |
|  | ''').show() |

[**view raw**](https://gist.github.com/dgadiraju/7406df149c6de20c81e413af4b2e4288/raw/a9f2c7d66f937cc73e98be2ba5c8258240571744/pyspark-sparksql-04-windowing.py)[**pyspark-sparksql-04-windowing.py**](https://gist.github.com/dgadiraju/7406df149c6de20c81e413af4b2e4288#file-pyspark-sparksql-04-windowing-py) hosted with  by [**GitHub**](https://github.com/)

## Problem Statement – Get top n products per day

Let us define the problem statement and see the real usage of analytics function.

* Problem Statement – Get top N Products Per day
* Get daily product revenue code from previous topic
* Use ranking functions and get the rank associated based on revenue for each day
* Once we get rank, let us filter for top n products.

## Understanding over, partition by and order by clauses

Let us understand different clauses required for analytics functions.

* Typical syntax – function(argument) over (partition by groupcolumn [order by [desc] ordercolumn])
* For aggregations we can define group by using partition by
* For ranking or windowing we need to use partition by and then order by. partition by is to group the data and order by is to sort the data to assign rank.
* We will not be able to use these any where except for select clause
* If we have to filter on these derived fields in select clause, we need to nest the whole query into another query.

## Performing aggregations

Let us see how to perform aggregations with in each group.

* We have functions such as sum, avg, min, max etc which can be used to aggregate the data.
* We need to use over (partition by) to get aggregations with in each group.
* Some realistic use cases
  + Get average salary for each department and get all employee details who earn more than average salary
  + Get average revenue for each day and get all the orders who earn revenue more than average revenue
  + Get highest order revenue and get all the orders which have revenue more than 75% of the revenue

## Using windowing functions

Let us see details about windowing functions with in each group

* We have functions such as lead, lag etc
* We need to use partition by and then order by for most of the windowing functions
* Some realistic use cases
  + Salary difference between current and next/previous employee with in each department

## Ranking with in each partition or group

Let us talk about ranking functions with in each group.

* We have functions like rank, dense\_rank, row\_number, first, last etc
* We need to use partition by and then order by for most of the windowing functions
* Some realistic use cases
  + Assign rank to employees based on salary with in each department
  + Assign ranks to products based on revenue each day or month

## Development Life Cycle

Let us talk about development life cycle.

* Take the DailyProductRevenue code which gives us order\_date, order\_item\_product\_id and revenue
* Add logic using function rank over partition by date and order by revenue in descending order. Make sure to give alias to this new field.
* Nest the query into another query – e. g.: select required\_fields from (query) query\_alias
* Add where clause on the query\_alias.derived field name
* Let us select and assign data related to order\_date, order\_item\_product\_id and revenue (after filtering on topN) to a new data frame
* Save the data frame into file

|  |  |
| --- | --- |
|  | [dev] |
|  | executionMode = local |
|  | input.base.dir = /Users/itversity/Research/data/retail\_db |
|  | output.base.dir = /Users/itversity/Research/data/bootcamp/pyspark |
|  |  |
|  | [prod] |
|  | executionMode = yarn-client |
|  | input.base.dir = /public/retail\_db |
|  | output.base.dir = /user/training/bootcamp/pyspark |

[**view raw**](https://gist.github.com/dgadiraju/a493ae53afd2db3d6526f0b3da64d372/raw/7bd470c424a9d6b2d772b901f8796b201df80d81/pyspark-sparksql-01-application.properties)[**pyspark-sparksql-01-application.properties**](https://gist.github.com/dgadiraju/a493ae53afd2db3d6526f0b3da64d372#file-pyspark-sparksql-01-application-properties) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | import configparser as cp, sys |
|  | from pyspark.sql import SparkSession |
|  |  |
|  | props = cp.RawConfigParser() |
|  | props.read('src/main/resources/application.properties') |
|  | env = sys.argv[1] |
|  |  |
|  | spark = SparkSession.\ |
|  | builder.\ |
|  | appName("Daily Product Revenue using Data Frames and Spark SQL").\ |
|  | master(props.get(env, 'executionMode')).\ |
|  | getOrCreate() |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | inputBaseDir = props.get(env, 'input.base.dir') |
|  | ordersCSV = spark.read. \ |
|  | csv(inputBaseDir + '/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | orderItemsCSV = spark.read. \ |
|  | csv(inputBaseDir + '/order\_items'). \ |
|  | toDF('order\_item\_id', 'order\_item\_order\_id', 'order\_item\_product\_id', |
|  | 'order\_item\_quantity', 'order\_item\_subtotal', 'order\_item\_product\_price') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  |  |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orderItems = orderItemsCSV.\ |
|  | withColumn('order\_item\_id', orderItemsCSV.order\_item\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_order\_id', orderItemsCSV.order\_item\_order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_product\_id', orderItemsCSV.order\_item\_product\_id.cast(IntegerType())). \ |
|  | withColumn('order\_item\_quantity', orderItemsCSV.order\_item\_quantity.cast(IntegerType())). \ |
|  | withColumn('order\_item\_subtotal', orderItemsCSV.order\_item\_subtotal.cast(FloatType())). \ |
|  | withColumn('order\_item\_product\_price', orderItemsCSV.order\_item\_product\_price.cast(FloatType())) |
|  |  |
|  | orders.createTempView('orders') |
|  | orderItems.createTempView('order\_items') |
|  |  |
|  | spark.conf.set('spark.sql.shuffle.partitions', '2') |
|  |  |
|  | dailyProductRevenue = spark.sql('''select o.order\_date, oi.order\_item\_product\_id, |
|  | round(sum(oi.order\_item\_subtotal), 2) as revenue |
|  | from orders o join order\_items oi |
|  | on o.order\_id = oi.order\_item\_order\_id |
|  | where o.order\_status in ("COMPLETE", "CLOSED") |
|  | group by o.order\_date, oi.order\_item\_product\_id |
|  | order by o.order\_date, revenue desc''') |
|  |  |
|  | dailyProductRevenue.createTempView('daily\_product\_revenue') |
|  |  |
|  | topNDailyProducts = spark.sql('''select q.order\_date, q.order\_item\_product\_id, q.revenue |
|  | from (select order\_date, order\_item\_product\_id, revenue, |
|  | rank() over (partition by order\_date order by revenue desc) rnk |
|  | from daily\_product\_revenue) q |
|  | where q.rnk <= %s |
|  | order by q.order\_date, q.revenue desc''' % topN) |
|  |  |
|  | outputBaseDir = props.get(env, 'output.base.dir') |
|  | topNDailyProducts.write.csv(outputBaseDir + '/topn\_daily\_products\_dfs') |

[**view raw**](https://gist.github.com/dgadiraju/a493ae53afd2db3d6526f0b3da64d372/raw/7bd470c424a9d6b2d772b901f8796b201df80d81/pyspark-sparksql-02-topN-daily-products.py)[**pyspark-sparksql-02-topN-daily-products.py**](https://gist.github.com/dgadiraju/a493ae53afd2db3d6526f0b3da64d372#file-pyspark-sparksql-02-topn-daily-products-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | spark-submit --master yarn \ |
|  | --deploy-mode client \ |
|  | --conf spark.ui.port=12901 \ |
|  | src/main/python/retail\_db/df/TopNDailyProductsDFS.py \ |
|  | prod 5 |

[**view raw**](https://gist.github.com/dgadiraju/a493ae53afd2db3d6526f0b3da64d372/raw/7bd470c424a9d6b2d772b901f8796b201df80d81/pyspark-sparksql-03-topN-daily-products.sh)[**pyspark-sparksql-03-topN-daily-products.sh**](https://gist.github.com/dgadiraju/a493ae53afd2db3d6526f0b3da64d372#file-pyspark-sparksql-03-topn-daily-products-sh) hosted with  by [**GitHub**](https://github.com/)

[**KAIZEN**](http://wip.itversity.com/)

# Different file formats and dealing with custom delimiters

posted on FEBRUARY 3, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

As part of this session we will talk about dealing with different file formats and also custom delimiters in text data. We will see how to read and how to write the data. Also we will understand APIs such as persist/cache on Data Frames.

* Overview of write APIs – dataframe.write
* Overview of read APIs – spark.read
* Supported file formats
  + csv, text (for text file formats)
  + json (using complex schema)
  + orc
  + parquet
  + avrò (3rd party)
* Processing text data with custom delimiters
* Persisting or Caching Data Frames

## Overview of write APIs – dataframe.write

<https://youtu.be/ICKs1ACqK8U>

Let us see how we can write data to different targets using APIs under write on top of data frame.

* Supported file formats – csv, text json, orc, parquet etc.
* We can also write data to 3rd party supported file formats such as avro
* Data can be written to Hive tables as well
* We can also connect to relational databases over JDBC and save our output into remote relational databases.
* We can also connect to any 3rd party database using relevant plugin and preserve data over there.

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.write. \ |
|  | format('json'). \ |
|  | save('/user/training/bootcampdemo/pyspark/orders\_json') |
|  |  |
|  | orders.write.json('/user/training/bootcampdemo/pyspark/orders\_json') |

[**view raw**](https://gist.github.com/dgadiraju/e04e98d0a4b118f64bcea62070b42309/raw/09888ef7c2e6981c19cdaf2291b58535092fc035/dataframe-write-examples-01-files.py)[**dataframe-write-examples-01-files.py**](https://gist.github.com/dgadiraju/e04e98d0a4b118f64bcea62070b42309#file-dataframe-write-examples-01-files-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | table = 'retail\_export.orders\_export' |
|  |  |
|  | orders.write. \ |
|  | format('jdbc'). \ |
|  | option('url', 'jdbc:mysql://ms.itversity.com'). \ |
|  | option('dbtable', 'retail\_export.orders\_export'). \ |
|  | option('user', 'retail\_user'). \ |
|  | option('password', 'itversity'). \ |
|  | save(mode='append') |
|  |  |
|  | orders.write. \ |
|  | jdbc("jdbc:mysql://ms.itversity.com", table, mode='append', |
|  | properties={"user": "retail\_user", |
|  | "password": "itversity"}) |

[**view raw**](https://gist.github.com/dgadiraju/e04e98d0a4b118f64bcea62070b42309/raw/09888ef7c2e6981c19cdaf2291b58535092fc035/dataframe-write-examples-02-jdbc.py)[**dataframe-write-examples-02-jdbc.py**](https://gist.github.com/dgadiraju/e04e98d0a4b118f64bcea62070b42309#file-dataframe-write-examples-02-jdbc-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | # To create new table and insert into it |
|  | orders.write. \ |
|  | format('hive'). \ |
|  | saveAsTable('bootcampdemo.orders\_hive', mode='overwrite') |
|  |  |
|  | orders.write.saveAsTable('bootcampdemo.orders\_hive', mode='overwrite') |
|  |  |
|  | # To insert data into existing table |
|  | orders.write. \ |
|  | format('hive'). \ |
|  | insertInto('bootcampdemo.orders\_hive', overwrite=True) |
|  |  |
|  | orders.write.insertInto('bootcampdemo.orders\_hive', overwrite=True) |
|  |  |
|  |  |

[**view raw**](https://gist.github.com/dgadiraju/e04e98d0a4b118f64bcea62070b42309/raw/09888ef7c2e6981c19cdaf2291b58535092fc035/dataframe-write-examples-03-hive.py)[**dataframe-write-examples-03-hive.py**](https://gist.github.com/dgadiraju/e04e98d0a4b118f64bcea62070b42309#file-dataframe-write-examples-03-hive-py) hosted with  by [**GitHub**](https://github.com/)

## Overview of read APIs – spark.read

spark.read have bunch of APIs to read data from different source types.

* Supported file formats- csv, text, json, orc, parquet etc
* We can also read data from 3rd party supported file formats such as avro
* We can read data directly from hive tables
* JDBC – to read data from relational databases
* There is generic API called format which can be used in conjunction with option to pass relevant arguments and then load data from either files or over JDBC.

|  |  |
| --- | --- |
|  | orders = spark.read. \ |
|  | format('json'). \ |
|  | load('/user/training/bootcampdemo/pyspark/orders\_json') |
|  |  |
|  | orders = spark.read.json('/user/training/bootcampdemo/pyspark/orders\_json') |
|  |  |
|  | orders.show() |
|  | orders.printSchema() |

[**view raw**](https://gist.github.com/dgadiraju/516b049c84babfda3be54ae5b0fa32b8/raw/8aa12c12eca67e0f241142bd71d9cb8c7d219874/dataframe-read-examples-01-files.py)[**dataframe-read-examples-01-files.py**](https://gist.github.com/dgadiraju/516b049c84babfda3be54ae5b0fa32b8#file-dataframe-read-examples-01-files-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | table = 'retail\_export.orders\_export' |
|  |  |
|  | orders = spark.read. \ |
|  | format('jdbc'). \ |
|  | option('url', 'jdbc:mysql://ms.itversity.com'). \ |
|  | option('dbtable', 'retail\_export.orders\_export'). \ |
|  | option('user', 'retail\_user'). \ |
|  | option('password', 'itversity'). \ |
|  | load() |
|  |  |
|  | orders = spark.read. \ |
|  | jdbc("jdbc:mysql://ms.itversity.com", table, |
|  | properties={"user": "retail\_user", |
|  | "password": "itversity"}) |
|  |  |
|  | orders.show() |
|  | orders.printSchema() |

[**view raw**](https://gist.github.com/dgadiraju/516b049c84babfda3be54ae5b0fa32b8/raw/8aa12c12eca67e0f241142bd71d9cb8c7d219874/dataframe-read-examples-02-jdbc.py)[**dataframe-read-examples-02-jdbc.py**](https://gist.github.com/dgadiraju/516b049c84babfda3be54ae5b0fa32b8#file-dataframe-read-examples-02-jdbc-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | orders = spark.read. \ |
|  | format('hive'). \ |
|  | table('bootcampdemo.orders\_hive') |
|  |  |
|  | orders = spark.read.table('bootcampdemo.orders\_hive') |
|  |  |
|  | orders.show() |
|  | orders.printSchema() |

[**view raw**](https://gist.github.com/dgadiraju/516b049c84babfda3be54ae5b0fa32b8/raw/8aa12c12eca67e0f241142bd71d9cb8c7d219874/dataframe-read-examples-03-hive.py)[**dataframe-read-examples-03-hive.py**](https://gist.github.com/dgadiraju/516b049c84babfda3be54ae5b0fa32b8#file-dataframe-read-examples-03-hive-py) hosted with  by [**GitHub**](https://github.com/)

## Supported file formats

Let us see details about all the supported formats in Spark to create data frames and save them.

* Following file formats are supported out of the box with Spark
  + text – using text (fixed length) or csv (delimited)
  + json
  + orc
  + parquet
* Avro is available with 3rd party plugins

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.selectExpr("concat(order\_id, ',', order\_date, ',', order\_customer\_id, ',', order\_status)"). \ |
|  | write. \ |
|  | format('text'). \ |
|  | save('/user/training/bootcampdemo/pyspark/orders\_text') |
|  |  |
|  | orders.selectExpr("concat(order\_id, ',', order\_date, ',', order\_customer\_id, ',', order\_status)"). \ |
|  | write. \ |
|  | text('/user/training/bootcampdemo/pyspark/orders\_text') |
|  |  |
|  | orders\_read = spark.read.format('text'). \ |
|  | load('/user/training/bootcampdemo/pyspark/orders\_text') |
|  |  |
|  | orders\_read = spark.read.text('/user/training/bootcampdemo/pyspark/orders\_text') |
|  |  |
|  | orders\_read.show() |
|  | orders\_read.printSchema() |

[**view raw**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40/raw/a409758cf4fdfbbc42a31eee0f39bc05bc5463cd/dataframe-file-formats-01-text.py)[**dataframe-file-formats-01-text.py**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40#file-dataframe-file-formats-01-text-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.write. \ |
|  | format('csv'). \ |
|  | save('/user/training/bootcampdemo/pyspark/orders\_csv') |
|  |  |
|  | orders.write.csv('/user/training/bootcampdemo/pyspark/orders\_csv') |
|  |  |
|  | orders\_read = spark.read. \ |
|  | format('csv'). \ |
|  | load('/user/training/bootcampdemo/pyspark/orders\_csv'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | orders\_read = spark.read. \ |
|  | csv('/user/training/bootcampdemo/pyspark/orders\_csv'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | orders\_read.show() |
|  | orders\_read.printSchema() |

[**view raw**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40/raw/a409758cf4fdfbbc42a31eee0f39bc05bc5463cd/dataframe-file-formats-02-csv.py)[**dataframe-file-formats-02-csv.py**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40#file-dataframe-file-formats-02-csv-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.write. \ |
|  | format('json'). \ |
|  | save('/user/training/bootcampdemo/pyspark/orders\_json') |
|  |  |
|  | orders.write.json('/user/training/bootcampdemo/pyspark/orders\_json') |
|  |  |
|  | orders\_read = spark.read. \ |
|  | format('json'). \ |
|  | load('/user/training/bootcampdemo/pyspark/orders\_json') |
|  |  |
|  | orders\_read = spark.read. \ |
|  | json('/user/training/bootcampdemo/pyspark/orders\_json') |
|  |  |
|  | orders\_read.show() |
|  | orders\_read.printSchema() |

[**view raw**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40/raw/a409758cf4fdfbbc42a31eee0f39bc05bc5463cd/dataframe-file-formats-03-json.py)[**dataframe-file-formats-03-json.py**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40#file-dataframe-file-formats-03-json-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.write. \ |
|  | format('orc'). \ |
|  | save('/user/training/bootcampdemo/pyspark/orders\_orc') |
|  |  |
|  | orders.write.orc('/user/training/bootcampdemo/pyspark/orders\_orc') |
|  |  |
|  | orders\_read = spark.read. \ |
|  | format('orc'). \ |
|  | load('/user/training/bootcampdemo/pyspark/orders\_orc') |
|  |  |
|  | orders\_read = spark.read. \ |
|  | orc('/user/training/bootcampdemo/pyspark/orders\_orc') |
|  |  |
|  | orders\_read.show() |
|  | orders\_read.printSchema() |

[**view raw**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40/raw/a409758cf4fdfbbc42a31eee0f39bc05bc5463cd/dataframe-file-formats-04-orc.py)[**dataframe-file-formats-04-orc.py**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40#file-dataframe-file-formats-04-orc-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.write. \ |
|  | format('parquet'). \ |
|  | save('/user/training/bootcampdemo/pyspark/orders\_parquet') |
|  |  |
|  | orders.write.parquet('/user/training/bootcampdemo/pyspark/orders\_parquet') |
|  |  |
|  | orders\_read = spark.read. \ |
|  | format('parquet'). \ |
|  | load('/user/training/bootcampdemo/pyspark/orders\_parquet') |
|  |  |
|  | orders\_read = spark.read. \ |
|  | parquet('/user/training/bootcampdemo/pyspark/orders\_parquet') |
|  |  |
|  | orders\_read.show() |
|  | orders\_read.printSchema() |

[**view raw**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40/raw/a409758cf4fdfbbc42a31eee0f39bc05bc5463cd/dataframe-file-formats-05-parquet.py)[**dataframe-file-formats-05-parquet.py**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40#file-dataframe-file-formats-05-parquet-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | # Launch pyspark with avro dependencies |
|  | # pyspark --master yarn --conf spark.ui.port=12901 --packages com.databricks:spark-avro\_2.11:4.0.0 |
|  |  |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.write. \ |
|  | format('com.databricks.spark.avro'). \ |
|  | save('/user/training/bootcampdemo/pyspark/orders\_avro') |
|  |  |
|  | orders\_read = spark.read. \ |
|  | format('com.databricks.spark.avro'). \ |
|  | load('/user/training/bootcampdemo/pyspark/orders\_avro') |
|  |  |
|  | orders\_read.show() |
|  | orders\_read.printSchema() |

[**view raw**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40/raw/a409758cf4fdfbbc42a31eee0f39bc05bc5463cd/dataframe-file-formats-06-avro.py)[**dataframe-file-formats-06-avro.py**](https://gist.github.com/dgadiraju/045a32545009ec5bd2a5bbf07c0cef40#file-dataframe-file-formats-06-avro-py) hosted with  by [**GitHub**](https://github.com/)

## Processing text data with custom delimiters

Now let us understand how to process text data with different line as well as field delimiters.

* We can read text data into RDD using SparkContext’s textFile. It will treat new line character as record delimiter.
* We have to parse each record in RDD and derive data to process further
* With Spark Data Frames we have csv and text APIs to read text data int Data Frame
* Both of them use new line character as record delimiter. When we use csv API to create data frame we can also specify field separator/delimiter using sep keyword argument
* We can also specify sep while writing data into text files with any field separator or delimiter using csv API. Also we can concatenate data as part of selectExpr with delimiter of our choice and use text API.
* Here is the example to read and write data with ascii null character.

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.selectExpr("concat(order\_id, '\00', order\_date, '\00', order\_customer\_id, '\00', order\_status)"). \ |
|  | write. \ |
|  | text('/user/training/bootcampdemo/pyspark/orders\_null') |
|  |  |
|  | orders.write.csv('/user/training/bootcampdemo/pyspark/orders\_null', '\00') |
|  |  |
|  | orders\_read\_csv = spark.read.csv('/user/training/bootcampdemo/pyspark/orders\_null', sep='\00'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | orders\_read = orders\_read\_csv. \ |
|  | withColumn('order\_id', orders\_read\_csv.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', orders\_read\_csv.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders\_read.show() |
|  | orders\_read.printSchema() |

[**view raw**](https://gist.github.com/dgadiraju/43e2324806cf494fa2379590b82cfd49/raw/c7bfaa6159152eaacde3992cdee3470740481f62/dataframe-custom-field-delimiter.py)[**dataframe-custom-field-delimiter.py**](https://gist.github.com/dgadiraju/43e2324806cf494fa2379590b82cfd49#file-dataframe-custom-field-delimiter-py) hosted with  by [**GitHub**](https://github.com/)

* At times, we might have to deal with text data where line delimiter is different than new line character.
* In this case we need to use HDFS APIs to read data from files with custom line delimiter into RDD and process further (either using transformations/actions or data frame operations)

|  |  |
| --- | --- |
|  | path = "/public/yelp-dataset/yelp\_review.csv" |
|  |  |
|  | yelpReview = sc.newAPIHadoopFile(path, |
|  | 'org.apache.hadoop.mapreduce.lib.input.TextInputFormat', |
|  | 'org.apache.hadoop.io.LongWritable', |
|  | 'org.apache.hadoop.io.Text', |
|  | conf={'textinputformat.record.delimiter' : '\r'}) |
|  |  |
|  | yelpReview.count() |
|  |  |
|  | for i in yelpReview.map(lambda r: str(r[1])).take(10): print(i) |
|  |  |
|  | for i in yelpReview. \ |
|  | map(lambda r: (len(str(r[1]).split('","')), 1)). \ |
|  | reduceByKey(lambda x, y: x + y). \ |
|  | collect(): |
|  | print(i) |

[**view raw**](https://gist.github.com/dgadiraju/bba062f110c7fd05815055c9caa7191d/raw/d676601f25c4bd0f65cfa57ab19c4ede312c2565/pyspark-custom-line-delimiters.py)[**pyspark-custom-line-delimiters.py**](https://gist.github.com/dgadiraju/bba062f110c7fd05815055c9caa7191d#file-pyspark-custom-line-delimiters-py) hosted with  by [**GitHub**](https://github.com/)

## Persisting or Caching Data Frames

Now let us see how we can persist data frames.

* By default data will be streamed as data frames to executor tasks as data being processed.
* Here is what will happen when data is read into executor task while it is being processed
  + Deserialize into object
  + Stream into memory
  + Process data by executor task by applying logic
  + Flush deserialized objects from memory as executor tasks are terminated
* Some times we might have to read same data multiple times for processing with in the same job. By default every time data need to be deserialized and submitted to executor tasks for processing
* To avoid deserializing into java objects when same data have to be read multiple times we can leverage caching.
* There are 2 methods persist and cache. By default with data frames caching will be done as MEMORY\_AND\_DISK from Spark 2.
* cache is shorthand method for persist at MEMORY\_AND\_DISK
* This is what happens when we cache Data Frame
  + Caching will be done only when data is read at least once for processing
  + Each record will be deserialized into object
  + These deserialized objects will be cached in memory as long as they fit
  + If not, deserialized objects will be spilled out to disk
* You can get details about different persistence levels from [here](https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.StorageLevel).

[← Previous Topic](http://wip.itversity.com/topic/spark-sql-analytics-functions-or-windowing-functions/)

# Compression Concepts and Algorithms

posted on FEBRUARY 3, 2019

**Topic Progress:** 

[← Back to Lesson](http://wip.itversity.com/lessons/building-spark-based-applications-using-python/)

As part of this topic we will understand compression algorithms and how we can actually compress data while saving output in particular file format.

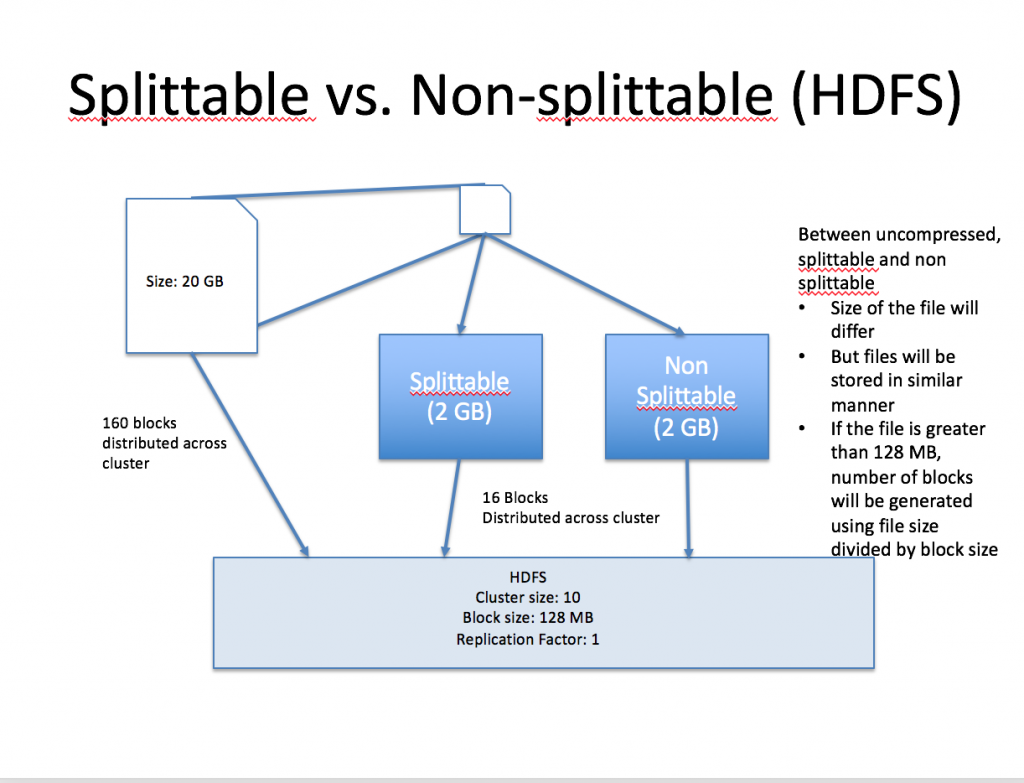
* Compression Algorithms
* Reading Compressed Data
* Compressing while saving output
* Criteria for choosing compression algorithm

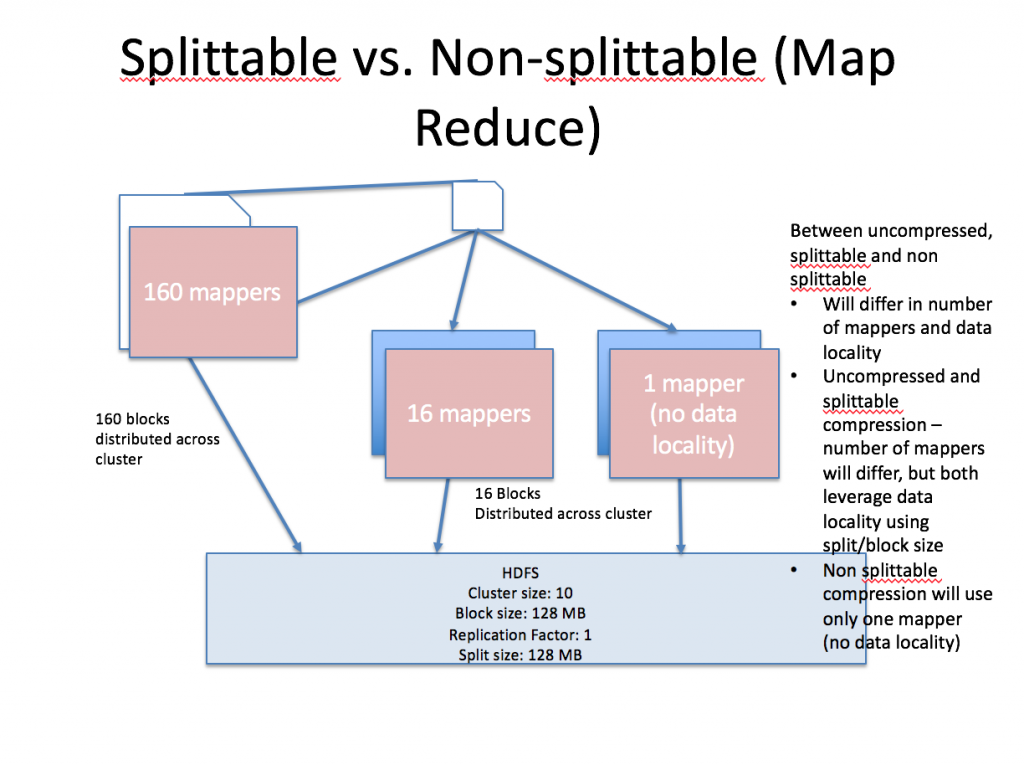
## Compression Algorithms

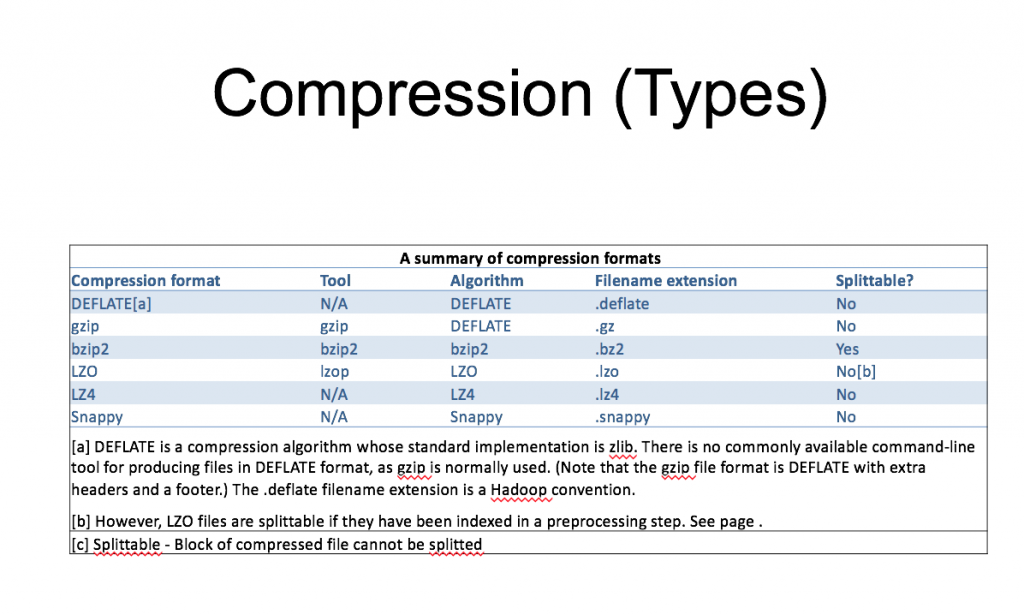
Let us understand details with respect to compression algorithms.

* Standard Algorithms – gzip, snappy, lzo, bzip2 etc
* Some of the compression algorithms are splittable while others are not.
* Most of the algorithms have both native implementation as well as java implementations (except bzip2 – which have only Java implementation)
* Native implementations are relatively faster than Java implementations
* Splittable vs. Non Splittable
* We can not only compress final output, but also intermediate data in Spark.

<https://youtu.be/BQvOO4_GYjw>

[](https://kaizen.itversity.com/wp-content/uploads/2018/08/SplittableVsNonSplittableHDFS.png)

[](https://kaizen.itversity.com/wp-content/uploads/2018/08/SplittableVsNonSplittableMapReduce.png)

[](https://kaizen.itversity.com/wp-content/uploads/2018/08/CompressionTypesOverview.png)

## Compression – Reading and Writing

* Compressing text files
  + Reading – No special action need to be taken as long as we use supported algorithms.
  + Writing
    - Can compress to most of the algorithms (bzip2, deflate, uncompressed, lz4, gzip, snappy, none)
    - Use option on spark.write before csv – df.write.option("codec", "gzip").csv("<PATH>")
    - Also option with compression work fine
* Compressing json files
  + Reading – No special action need to be taken as long as we use supported algorithms.
  + Writing
    - Can compress to most of the algorithms (bzip2, deflate, uncompressed, lz4, gzip, snappy, none)
    - Use option with compression – option("compression", "gzip")
* Compressing orc files
  + Reading – No special action need to be taken as long as we use supported algorithms.
  + Writing
    - Default – snappy
    - Could not figure out how I can write in other file formats
* Compressing parquet files
  + Reading – No special action need to be taken as long as we use supported algorithms.
  + Writing
    - Default – snappy
    - Supported codecs – uncompressed, snappy, gzip, lzo
    - Set spark.sql.parquet.compression.codec to the appropriate algorithm
* Compressing avrò files
  + Reading – No special action need to be taken as long as we use supported algorithms.
  + Writing
    - Default – uncompress
    - Supported codecs – uncompressed, snappy, deflate
    - Set spark.sql.avro.compression.codec to the appropriate algorithm

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.write.option("codec", "gzip"). \ |
|  | csv("/user/training/bootcampdemo/pyspark/orders\_csv\_compressed") |
|  |  |
|  | orders.write. \ |
|  | format('csv'). \ |
|  | option("codec", "gzip"). \ |
|  | save("/user/training/bootcampdemo/pyspark/orders\_csv\_compressed", mode='overwrite') |
|  |  |
|  | orders.selectExpr("concat(order\_id, ',', order\_date, ',', order\_customer\_id, ',', order\_status)"). \ |
|  | write.option("codec", "gzip"). \ |
|  | text("/user/training/bootcampdemo/pyspark/orders\_text\_compressed") |

[**view raw**](https://gist.github.com/dgadiraju/2184c6aa9fa78c70425ade2aa537a0b3/raw/bd2949532a8a038c77e4ffeca5d2eff94988e9cb/dataframe-file-formats-compression-01-text.py)[**dataframe-file-formats-compression-01-text.py**](https://gist.github.com/dgadiraju/2184c6aa9fa78c70425ade2aa537a0b3#file-dataframe-file-formats-compression-01-text-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | orders.write.option("codec", "gzip"). \ |
|  | json("/user/training/bootcampdemo/pyspark/orders\_json\_compressed") |
|  |  |
|  | orders.write. \ |
|  | format('json'). \ |
|  | option("codec", "gzip"). \ |
|  | save("/user/training/bootcampdemo/pyspark/orders\_json\_compressed", mode='overwrite') |

[**view raw**](https://gist.github.com/dgadiraju/2184c6aa9fa78c70425ade2aa537a0b3/raw/bd2949532a8a038c77e4ffeca5d2eff94988e9cb/dataframe-file-formats-compression-02-json.py)[**dataframe-file-formats-compression-02-json.py**](https://gist.github.com/dgadiraju/2184c6aa9fa78c70425ade2aa537a0b3#file-dataframe-file-formats-compression-02-json-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | spark.conf.set('spark.sql.parquet.compression.codec', 'gzip') |
|  |  |
|  | orders.write. \ |
|  | format('parquet'). \ |
|  | save('/user/training/bootcampdemo/pyspark/orders\_parquet\_compressed') |
|  |  |
|  | orders.write.parquet('/user/training/bootcampdemo/pyspark/orders\_parquet', mode='overwrite') |

[**view raw**](https://gist.github.com/dgadiraju/2184c6aa9fa78c70425ade2aa537a0b3/raw/bd2949532a8a038c77e4ffeca5d2eff94988e9cb/dataframe-file-formats-compression-03-parquet.py)[**dataframe-file-formats-compression-03-parquet.py**](https://gist.github.com/dgadiraju/2184c6aa9fa78c70425ade2aa537a0b3#file-dataframe-file-formats-compression-03-parquet-py) hosted with  by [**GitHub**](https://github.com/)

|  |  |
| --- | --- |
|  | ordersCSV = spark.read.csv('/public/retail\_db/orders'). \ |
|  | toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status') |
|  |  |
|  | from pyspark.sql.types import IntegerType, FloatType |
|  | orders = ordersCSV. \ |
|  | withColumn('order\_id', ordersCSV.order\_id.cast(IntegerType())). \ |
|  | withColumn('order\_customer\_id', ordersCSV.order\_customer\_id.cast(IntegerType())) |
|  |  |
|  | spark.conf.set('spark.sql.avro.compression.codec', 'snappy') |
|  |  |
|  | orders.write. \ |
|  | format('com.databricks.spark.avro'). \ |
|  | save('/user/training/bootcampdemo/pyspark/orders\_avro\_compressed') |

[**view raw**](https://gist.github.com/dgadiraju/2184c6aa9fa78c70425ade2aa537a0b3/raw/bd2949532a8a038c77e4ffeca5d2eff94988e9cb/dataframe-file-formats-compression-04-avro.py)[**dataframe-file-formats-compression-04-avro.py**](https://gist.github.com/dgadiraju/2184c6aa9fa78c70425ade2aa537a0b3#file-dataframe-file-formats-compression-04-avro-py) hosted with  by [**GitHub**](https://github.com/)

## Criteria and Tips

Here is the criteria and tips for choosing the compression algorithms.

* Choose the ones with native implementation
* Most of those compression algorithms which have native implementations are not splittable (which means irrespective of the size of the file, each file is processed by one task at a time).
* To work around the limitation of one task per file in case of non splittable algorithms we need to make sure data is saved in multiple files of manageable size.
* Some of the file formats such as parquet, orc etc are compressed by default. It is better to use the default compression (for example parquet is compressed using snappy).