SQL query Execution flow

from -> where and/or join -> group by -> having -> select -> order by

spark work flow

* Using spark-submit, the user submits an application.
* In spark-submit, we invoke the main() method that the user specifies. It also launches the driver program.
* The driver program asks for the resources to the cluster manager that we need to launch executors.
* The cluster manager launches executors on behalf of the driver program.
* The driver process runs with the help of user application. Based on the [**actions and transformation on RDDs**](https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/), the driver sends work to executors in the form of tasks.
* The executors process the task and the result sends back to the driver through the cluster manager.

Apache **Spark** is an in-memory distributed data processing engine and **YARN** is a cluster management technology. ... As Apache **Spark** is an in-memory distributed data processing engine, application performance is heavily dependent on resources such as executors, cores, and memory allocated

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

**RDD** – **RDD** is a distributed collection of data elements spread across many machines in the cluster. **RDDs** are a set of Java or Scala objects representing data. **DataFrame** – A **DataFrame** is a distributed collection of data organized into named columns. It is conceptually equal to a table in a relational database.

Spark Modules

Core – Transformations and actions

Spark SQL and Data Frames

Structured Streaming

Machine learning Pipelines

GraphX Pipelines

Spark Data Structures

Resilient Distributed datasets (an in memory distributed collection)

Data Frame (a wrapper on top of RDD with structure)

Spark Framework And Execution Modes

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

Diff executor mode supported by spark are

1. Local(for development)
2. Standalone(for development)
3. Mesos (production)
4. YARN (production)

--------------------

Data frames is nothing but collections with structures

[] echo $SPARK\_MAJOR\_VERSION

Launch pyspark on linux shell ---------------🡪 pyspark --master yarn –conf spark.ui.port=12901 --num-executors 2

🡪object sc it is a spark context

\*\*orderItems = sc.textFile(“Hdfs path”)

In this case spark integrated with hadoop give hdfs path

>>> orderItems = sc.textFile("/user/abhilashkalva/data/retail\_db/order\_items.txt")

>>> orderItems.take(10)

It will print first 10 records

>>> type(orderItems)

<class 'pyspark.rdd.RDD'> ---which is similar to list (Rdd is a Distributed dataset/collection where as list is a liner collection)

>>> orderItems

/user/abhilashkalva/data/retail\_db/order\_items.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:-2

>>> help(orderItems) ----shows the list of api’s

>>> orderItemsFiltered = orderItems.filter(lambda oi : int(oi.split(',')[1])==2)

>>> orderItemsFiltered.collect()

[u'2,2,1073,1,199.99,199.99', u'3,2,502,5,250.0,50.0', u'4,2,403,1,129.99,129.99']

>>> orderItemsMap = orderItemsFiltered.map(lambda oi : float(oi.split(',')[4]))

>>> orderItemsMap.collect()

[199.99, 250.0, 129.99]

>>> orderItemsMap.reduce(lambda x,y:x+y)

579.98

>> orderItems =

spark(or sqlContext).read.json("/user/abhilashkalva/data/data/retail\_db\_json/order\_items")

It create a data structure called dataFrame.

DataFrame nothing but a RDD with structure. internally everything is Rdd with respective spark but DataFrame is wrapper on top of typical Rdd where a structure is on defined on it

>>> type(orderItems)

<class 'pyspark.sql.dataframe.DataFrame'>

>>> orderItems.groupBy("order\_items\_order\_id")

<pyspark.sql.group.GroupedData object at 0x725ab434b9d0>

>>> orderItems.filter("order\_item\_order\_id = 2").show()

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

|order\_item\_id|order\_item\_order\_id|order\_item\_product\_id|order\_item\_product\_price|order\_item\_quantity|order\_item\_subtotal|

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

| 2| 2| 1073| 199.99| 1| 199.99|

| 3| 2| 502| 50.0| 5| 250.0|

| 4| 2| 403| 129.99| 1| 129.99|

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

The Api’s such as map reduce are considered to be core api’s or core modules

When It comes to core api’s we use Rdd when it comes to on sparks sql and DataFrame operations we use DataFrame

Spark is also called combination of the jar files, configuration files which contains the properties controls the runtime behaviour of spark

[abhilashkalva@gw02 ~]$ cd /usr/hdp/2.5.0.0-1245/spark

[abhilashkalva@gw02 spark]$ ls -ltr

total 16

drwxr-xr-x 2 root root 4096 Aug 25 2016 aux

drwxr-xr-x 2 spark spark 4096 Jun 5 2017 logs

drwxr-xr-x 3 root root 4096 Oct 8 2018 python

drwxr-xr-x 2 root root 4096 Oct 8 2018 lib

[abhilashkalva@gw02 spark]$ cd lib

[abhilashkalva@gw02 lib]$ ls -ltr

total 2292

-rw-r--r-- 1 root root 2052514 Jun 19 2016 spark-streaming\_2.10-1.6.2.jar

-rw-r--r-- 1 root root 86044 Feb 12 2018 spark-streaming-flume-sink\_2.10-1.6.2.jar

-rw-r--r-- 1 root root 103386 Feb 14 2018 spark-streaming-flume\_2.10-1.6.2.jar

-rw-r--r-- 1 root root 93757 Apr 30 2018 spark-avro\_2.10-2.0.1.jar

Here we can see there is a bunch of jar files which support the spark framework so when develop spark jobs and when submit the spark jobs it will use these binaries to run spark job on cluster

[abhilashkalva@gw02 ~]$ cd /etc/spark2/conf

[abhilashkalva@gw02 conf]$ ls -ltr

total 68

-rwxr-xr-x 1 root root 3861 Aug 25 2016 spark-env.sh.template

-rw-r--r-- 1 root root 1292 Aug 25 2016 spark-defaults.conf.template

-rw-r--r-- 1 root root 865 Aug 25 2016 slaves.template

-rw-r--r-- 1 root root 7239 Aug 25 2016 metrics.properties.template

-rw-r--r-- 1 root root 2025 Aug 25 2016 log4j.properties.template

-rw-r--r-- 1 root root 1105 Aug 25 2016 fairscheduler.xml.template

-rw-r--r-- 1 root root 987 Aug 25 2016 docker.properties.template

-rw-r--r-- 1 spark spark 4956 Jun 5 2017 metrics.properties

-rwxr-xr-x 1 spark spark 244 Jun 5 2017 spark-thrift-fairscheduler.xml

-rw-r--r-- 1 spark spark 6178 Jun 3 2019 hbase-site.xml

-rw-r--r-- 1 hive hadoop 1002 Jun 25 2019 spark-thrift-sparkconf.conf

-rw-r--r-- 1 spark spark 1817 Mar 11 2020 spark-env.sh

lrwxrwxrwx 1 root root 28 Oct 5 01:07 hive-site.xml -> /etc/hive/conf/hive-site.xml

-rw-r--r-- 1 spark spark 1240 Oct 24 20:31 log4j.properties

-rw-r--r-- 1 spark spark 900 Dec 14 02:32 spark-defaults.conf

-🡪property files here

So if u go through property files their will be bunch of properties which will control the runtime behaviour of spark jobs

[abhilashkalva@gw02 conf]$ view spark-env.sh

[abhilashkalva@gw02 conf]$ view spark-defaults.conf

------🡪

[abhilashkalva@gw02 conf]$ pyspark --master yarn --conf spark.ui.port=12901 --num-executors 2 --conf spark.dynamicAllocation.enabled=false

When we launch pyspark with these properties we can actually access the web service associated with it using this web ui 🡪 http://rm01.itversity.com:19088/cluster

<http://rm01.itversity.com:19088/proxy/application_1607930259925_1925/>

pyspark --master yarn

🡪we have used yarn. so it will be running in yarn mode one of the execution modes of spark

--num-executors 2

🡪their are 2 worker nodes and 2 executers

Driver is not executer. actual process data in this driver

<http://rm01.itversity.com:19088/proxy/application_1607930259925_1925/executors/>

executor is nothing but a JVM

sparkContext:

which Is nothing but a web service which will manage those resources what ever created for the session..each web service run on a different port number (10000 to 65535)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

[abhilashkalva@gw02 conf]$ pyspark --master yarn --conf spark.ui.port=12901 …..

Is a Driver program when it is launched it has created the sparkContext. sparkContext it actually

Procured some resources from the clusters by default 2 executors and then sparkContext will keep track of it.

\*\*\*\*\*\*\*\*\*\*\*\*

Flat map:

Flat map is an api which needs to be use which can convert each line in to individual words as records and then we have to perform count on those words

What is flatMap in PySpark?

**PySpark flatMap**() is a transformation operation that flattens the RDD/DataFrame (array/map DataFrame columns) after applying the function on every element and returns a new **PySpark** RDD/DataFrame

>>> words = lines.flatMap(lambda line: line.split(" "))

>>> words.collect()

[u'hi', u'my', u'name', u'is', u'abhilash.iam', u'use', u'flat', u'map', u'to', u'count', u'words', u'on', u'this', u'text', u'file.', u'we', u'can', u'see', u'what', u'happens']

>>> words.count()

20

>>> wordTuples = words.map(lambda word :(word,1))

>>> type(wordTuples)

<class 'pyspark.rdd.PipelinedRDD'> (because using existing rdd type –pipielineRDD)

-------🡪reduceByKey : the diff b/w reduce and reduce by key is in this case reduce will be running each key (based on keys it will be grouping the data and it wiil be perform this arithmetic operations on top of values for each grouped key )

>>> wordCount.saveAsTextFile("/user/abhilashkalva/books/wordscount")

[abhilashkalva@gw02 ~]$ hadoop fs -ls /user/abhilashkalva/books/wordscount

Found 3 items

-rw-r--r-- 2 abhilashkalva hdfs 0 2020-12-22 10:28 /user/abhilashkalv a/books/wordscount/\_SUCCESS

-rw-r--r-- 2 abhilashkalva hdfs 127 2020-12-22 10:28 /user/abhilashkalv a/books/wordscount/part-00000

-rw-r--r-- 2 abhilashkalva hdfs 131 2020-12-22 10:28 /user/abhilashkalv a/books/wordscount/part-00001

23-12-2020

Apache Spark 2 - RDD, DAG and Lazy Evaluation

https://spark.apache.org/docs/latest/rdd-programming-guide.html

Data structures – RDD and Data Frames

Quick ovewies about Api’s -- Transformations and actions

Direct Acyclic Graph and lazy Evalution

>>> sc.setLogLevel("INFO")

Remove log level info use

>>> sc.setLogLevel("ERROR")

>>> lines = sc.textFile("/public/randomtextwriter/part-m-00000")

>>> type(lines)

<class 'pyspark.rdd.RDD'>

When ever transformation is performed a new DAG will be created

Resilient Distributed Datasets (**RDD**) is a fundamental data structure of **Spark**. It is an immutable distributed collection of objects. Each dataset in **RDD** is divided into logical partitions, which may be computed on different nodes of the cluster. ... Formally, an **RDD** is a read-only, partitioned collection of records.

All the hdfs blocks considered as rdd partitions.

It will just read the data in streaming fashion and then output will be saved into memory as available if not it is saved into someware else

Resilent Distributed Datasets is the fundamental data structure in spark

* In-memory
* Distributed
* Resilent
* Data from files will be divided into RDD partitions and each partition processed by separate task
* By default it will use HDFS block size (128 MB) to determine partition
* We can increase(cannot decrease) number of partitions by using additional parameter in sc.textFile
* By default when data is loaded into memory each record will be serialized into java object
* We can persist the RDD partitions at different storage levels
* Memory\_ONLY(default)
* MEMORY\_AND\_DISK
* DISK\_ONLY

\*\*Rdd is nothing but serialized java objects of input data

\*\*Rdd partition is nothing but serialized java objects for a given block

\*\* block is a part of text data

\*\*RDD is nothing but bunch of partitions and each rdd partition hava serialized java object which are created by reading data from file into memory

And then once the first rdd is created once it is process they will be rotating as the data is being processed and finally again those java objectswill be deserialized and will be return into file.

1 gb of text data is divided into eight 128mb blocks AND one 28mb block

[abhilashkalva@gw02 ~]$ hdfs fsck /public/randomtextwriter/part-m-00000 -files - blocks

Connecting to namenode via http://172.16.1.101:50070/fsck?ugi=abhilashkalva&file s=1&blocks=1&path=%2Fpublic%2Frandomtextwriter%2Fpart-m-00000

FSCK started by abhilashkalva (auth:SIMPLE) from /172.16.1.109 for path /public/ randomtextwriter/part-m-00000 at Wed Dec 23 04:17:54 EST 2020

/public/randomtextwriter/part-m-00000 1102230331 bytes, 9 block(s): OK

0. BP-292116404-172.16.1.101-1479167821718:blk\_1074171511\_431441 len=134217728 r epl=3

1. BP-292116404-172.16.1.101-1479167821718:blk\_1074171524\_431454 len=134217728 r epl=3

2. BP-292116404-172.16.1.101-1479167821718:blk\_1074171559\_431489 len=134217728 r epl=3

3. BP-292116404-172.16.1.101-1479167821718:blk\_1074171609\_431539 len=134217728 r epl=3

4. BP-292116404-172.16.1.101-1479167821718:blk\_1074171657\_431587 len=134217728 r epl=3

5. BP-292116404-172.16.1.101-1479167821718:blk\_1074171691\_431621 len=134217728 r epl=3

6. BP-292116404-172.16.1.101-1479167821718:blk\_1074171721\_431651 len=134217728 r epl=3

7. BP-292116404-172.16.1.101-1479167821718:blk\_1074171731\_431661 len=134217728 r epl=3

8. BP-292116404-172.16.1.101-1479167821718:blk\_1074171736\_431666 len=28488507 re pl=3

Status: HEALTHY

Status: HEALTHY

Total size: 1102230331 B

Total dirs: 0

Total files: 1

Total symlinks: 0

Total blocks (validated): 9 (avg. block size 122470036 B)

Minimally replicated blocks: 9 (100.0 %)

Over-replicated blocks: 0 (0.0 %)

Under-replicated blocks: 0 (0.0 %)

Mis-replicated blocks: 0 (0.0 %)

Default replication factor: 2

Average block replication: 3.0

Corrupt blocks: 0

Missing replicas: 0 (0.0 %)

Number of data-nodes: 5

Number of racks: 1

FSCK ended at Wed Dec 23 04:17:54 EST 2020 in 1 milliseconds

Compiled code is copied into the cache of executor and the depending upon the blocks the tasks will be created

Job is nothing but when ever the action is performed it will result in a job and job will have multiple stages depending upon the complexity of the logic.

Important concepts of Rdd:-

RDD persistence:

>>> lines = sc.textFile("/public/randomtextwriter/part-m-00000")

>>> from pyspark import StorageLevel

>>> lines.persist(StorageLevel.MEMORY\_ONLY)

/public/randomtextwriter/part-m-00000 MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0

>>> lines.cache()

/public/randomtextwriter/part-m-00000 MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0

>>> lines.count()

26421

Once the data is persisted reprocess the data again and again

Once the sparkContext is close what ever data persisted is gone

Data Frame:-

Data frame is nothing but Structure on top of RDD

Many time data will have structure. Using RDD and then core API’s is some what tedious and criptic. 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 on statistics\_collected

Transformations And Actions :-

* Transformations : which take a RDD and return another RDD as output.

Input is an RDD and out put also an RDD.these APIs does not trigger execution but update the DAG.

* Row level Transformations - map, flatMap, filter
* Joins – join,leftOuterJoin, rightOuterJoin
* Aggregations – reduceByKey, aggregateByKey
* Sorting data – sortByKey
* Group operations such as groupByKey
* Set operations – union, intersection
* And more
* Actions:- 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 triggers execution of DAG.
* Previewing data – first, take, takeSample
* Converting RDD into typical collections – collect
* Total Aggregations – count, reduce
* Total ranking – top
* Saving files – saveAsTextFile, saveAsNewAPIHadoopFile etc
* And more

In Scala --- lines.toDebugString 🡪 we can see DAG level details

Directed Acyclic Graph(DAG) and Lazy Evaluation:-

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

* When we create a spark Context object it will procure resources in 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 variable 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 have a DAG associated with it.
* When APIs which are categorized as action (such as take, collect, saveAsTextFile) are used Dag associated with the variable executed.
* In Scala, we can look at Dag details by using toDebugString on top the variable created.
* We can visualize the DAG as part of spark.

25-12-2020

Apache Spark 2 - Basic Transformations and Actions - 01 - map, flatMap, reduce and more

>>> l =list(range(1,100001))

>>> type(l)

<type 'list'>

>>> len(l)

100000

# Convert a typical collection type to RDD use sc.parallelize()

# conver a text file to RDD use sc.textFile()

>>> lRDD = sc.parallelize(l)

>>> type(lRDD)

<class 'pyspark.rdd.RDD'>

>>> lRDD.count()

20/12/24 03:11:08 WARN TaskSetManager: Stage 0 contains a task of very large size (155 KB). The maximum recommended task size is 100 KB.

100000

>>>lRDD.first() # 1

>>>lRDD.take(10) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# RDD to collection use lRDD.collect() 🡪 all data loaded into a Serialized collection of type list.

To get sum of even numbers in 100000 records

>>> lEven = lRDD.filter(lambda x : x%2 == 0)

>>> type(lEven)

<class 'pyspark.rdd.PipelinedRDD'>

>>> lEven.count()

20/12/24 03:26:07 WARN TaskSetManager: Stage 4 contains a task of very large size (155 KB). The maximum recommended task size is 100 KB.

50000

>>> lEven.reduce(lambda x,y : x+y)

20/12/24 03:29:08 WARN TaskSetManager: Stage 5 contains a task of very large size (155 KB). The maximum recommended task size is 100 KB.

2500050000

Or

>>> from operator import add

>>> lEven.reduce(add)

20/12/24 04:04:39 WARN TaskSetManager: Stage 6 contains a task of very large size (155 KB). The maximum recommended task size is 100 KB.

2500050000

Word Count

>>> l = ["Hello How are you?", "yuu are welcome" , "welcome to tandur"]

>>> len(l)

3

>>> lRDD.count()

3

As long as the lambda functions the return value is collection flatMap will flatten this out into multiple records.

>>> lFlatMap = lRDD.flatMap(lambda s : s.split(" "))

>>> lFlatMap.count()

10

>>> lFlatMap.collect()

['Hello', 'How', 'are', 'you?', 'yuu', 'are', 'welcome', 'elcome', 'to', 'tandur']

>>> lRDD.count()

3

We will get all 10 record in separate lines

>>> for i in lFlatMap.collect(): print(i)

...

Hello

How

are

you?

yuu

are

welcome

to

tandur

these records can be passed into reduceByKey and we can get the count.

>> lMap = lFlatMap.map(lambda s: (s,1))

>>> lMap.count()

10

>>> lMap.collect()

[('Hello', 1), ('How', 1), ('are', 1), ('you?', 1), ('yuu', 1), ('are', 1), ('welcome', 1), ('elcome', 1), ('to', 1), ('tandur', 1)]

reduceByKey is a transformation. Reduce is a Action

>>> wc = lMap.reduceByKey(lambda x,y: x+y)

>>> from operator import add

>>> wc = lMap.reduceByKey(add)

>>> wc.count()

9

>>> wc.collect()

[('you?', 1), ('tandur', 1), ('yuu', 1), ('to', 1), ('welcome', 1), ('How', 1), ('are', 2), ('elcome', 1), ('Hello', 1)]

reduceByKey performing the Aggregation

>>> lines = sc.textFile("/public/randomtextwriter/part-m-00000")

>>> lines.count()

26421

>>> words = lines.flatMap(lambda s: s.split(" "))

>>> words.count()

100809099

>>> wordTuples = words.map(lambda word: (word,1))

>>> wordTuples.count()

>>> for i in wordTuples.take(10): print(i)

...

(u'SEQ\x06\x19org.apache.hadoop.io.Text\x19org.apache.hadoop.io.Text\x00\x00\x00\x00\x00\x00\ufffdg\x05\x081\ufffd\ufffdJ$\ufffd\u05d2\x1ad\ufffd\x08\x00\x00\x02\ufffd\x00\x00\x00XWpterostigma', 1)

(u'steprelationship', 1)

(u'pleasurehood', 1)

(u'abusiveness', 1)

(u'seelful', 1)

(u'unstipulated', 1)

(u'winterproof', 1)

(u'\ufffd\x02gmericarp', 1)

(u'pentosuria', 1)

(u'airfreighter', 1)

>>> wordCount = wordTuples.reduceByKey(add)

>>> wordCount.count()

1588631

>>>for i in wordCount.take(10): print(i)

>>> orders = sc.textFile("/public/retail\_db/orders")

>>> orders.first()

u'1,2013-07-25 00:00:00.0,11599,CLOSED'

>>> for i in orders.take(4): print(i)

...

1,2013-07-25 00:00:00.0,11599,CLOSED

2,2013-07-25 00:00:00.0,256,PENDING\_PAYMENT

3,2013-07-25 00:00:00.0,12111,COMPLETE

4,2013-07-25 00:00:00.0,8827,CLOSED

We want order\_status

We need to apply split using “,” as delimiter it will convert into array that array of 3 give the order status

>>> o = "1,2013-07-25 00:00:00.0,11599,CLOSED"

>>> o.split(",")[3]

'CLOSED'

>>> orders.map(lambda o: o.split(","))

PythonRDD[33] at RDD at PythonRDD.scala:48

>>> orders.map(lambda o: o.split(",")).take(4)

[[u'1', u'2013-07-25 00:00:00.0', u'11599', u'CLOSED'], [u'2', u'2013-07-25 00:00:00.0', u'256', u'PENDING\_PAYMENT'], [u'3', u'2013-07-25 00:00:00.0', u'12111', u'COMPLETE'], [u'4', u'2013-07-25 00:00:00.0', u'8827', u'CLOSED']]

>>> orders.map(lambda o: o.split(",")[3]).take(4)

[u'CLOSED', u'PENDING\_PAYMENT', u'COMPLETE', u'CLOSED']

We get the Year

>>> orders.map(lambda o: o.split(",")[1][:4]).take(4)

>>> orders.map(lambda o: int(o.split(",")[1][:4])).take(4)

Shuffling:

>>> orderItems = sc.textFile("/public/retail\_db/order\_items")

>>> type(orderItems)

<class 'pyspark.rdd.RDD'>

>>> for i in orderItems.take(8): print(i)

...

1,1,957,1,299.98,299.98

2,2,1073,1,199.99,199.99

3,2,502,5,250.0,50.0

4,2,403,1,129.99,129.99

5,4,897,2,49.98,24.99

6,4,365,5,299.95,59.99

7,4,502,3,150.0,50.0

8,4,1014,4,199.92,49.98

* Compute revenue for each order id

Each eliment in the RDD has to be topple

Tuple is a unnamed object

>>> orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.split(",")[4])))

>>> for i in orderItemsMap.take(8): print(i) ...

(1, 299.98)

(2, 199.99)

(2, 250.0)

(2, 129.99)

(4, 49.98)

(4, 299.95)

(4, 150.0)

(4, 199.92)

>>> orderItemsMapGBK = orderItemsMap.groupByKey()

>>> orderItemsMap.count()

172198

>>> orderItemsMapGBK.count()

57431

>>> for i in orderItemsMapGBK.take(3):print(i)

...

(2, <pyspark.resultiterable.ResultIterable object at 0x6a844f8b8710>)

(4, <pyspark.resultiterable.ResultIterable object at 0x6a844f8b8e90>)

(8, <pyspark.resultiterable.ResultIterable object at 0x6a844f8b8f10>)

Here value is a iterable

>>> orderRevenue = orderItemsMapGBK.map(lambda o: (o[0], sum(o[1])))

>>> for i in orderRevenue.take(5): print(i)

...

(2, 579.98)

(4, 699.85)

(8, 729.8399999999999)

(10, 651.9200000000001)

(12, 1299.8700000000001)

Or

from operator import add

>>> orderRevenueRBK = orderItemsMap.reduceByKey(add) / or

>>> orderRevenueRBK = orderItemsMap.reduceByKey(lambda x,y: x+y)

>>> for i in orderRevenueRBK.take(5): print(i)

...

(2, 579.98)

(4, 699.85)

(8, 729.8399999999999)

(10, 651.9200000000001)

(12, 1299.8700000000001)

For aggregations reduceByKey is best for Use

25-12-2020

Apache Spark 2 - Basic Transformations and Actions - 02 - shuffling and aggregations

>>> orderItems = sc.textFile("/public/retail\_db/order\_items")

>>> orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.s plit(",")[4])))

>>> type(orderItemsMap)

>>> orderItemsMap.take(5)

[(1, 299.98), (2, 199.99), (2, 250.0), (2, 129.99), (4, 49.98)]

groupBykey does not take any argument

>>> orderItemsGBK = orderItemsMap.groupByKey().take(3)

reduceBykey takes lambda as an argument

# sc.textFile -> map( 4, 49.98) (4, 299.95) (4, 150.0) (4, 199.92) 🡪 groupByKey (4, [49.98, 299.95, 150.0,199.92])

# sc.textFile 🡪 map(4,349.93) (4, 349.92) -> reduceByKey(4, 699.81)

Diff b/w groupByKey and reduceByKey :--

When a **groupByKey** is called on a RDD pair the data **in the** partitions are shuffled over the network to form a **key** and list of values. The **reduceByKey** works much better on a large dataset as compared to. That's because **Spark** knows it can combine output **with a** common **key** on each partition before shuffling the data.

>>> orderItems.repartition(4).saveAsTextFile("/user/abhilashkalva/pyspark/orderItemsPartitioned")

Data is partition into 4 buckets

[abhilashkalva@gw02 ~]$ hadoop fs -ls /user/abhilashkalva/pyspark/orderItemsPartitioned

Found 5 items

-rw-r--r-- 2 abhilashkalva hdfs 0 2020-12-25 04:29 /user/abhilashkalva/pyspark/orderItemsPartitioned/\_SUCCESS

-rw-r--r-- 2 abhilashkalva hdfs 1351889 2020-12-25 04:29 /user/abhilashkalva/pyspark/orderItemsPartitioned/part-00000

-rw-r--r-- 2 abhilashkalva hdfs 1352498 2020-12-25 04:29 /user/abhilashkalva/pyspark/orderItemsPartitioned/part-00001

-rw-r--r-- 2 abhilashkalva hdfs 1351882 2020-12-25 04:29 /user/abhilashkalva/pyspark/orderItemsPartitioned/part-00002

-rw-r--r-- 2 abhilashkalva hdfs 1352611 2020-12-25 04:29 /user/abhilashkalva/pyspark/orderItemsPartitioned/part-00003

Partitioning and grouping based on the no of partitions that needs to be used to perform these operations such as groupByKey which data is shuffle the data will be partitioned and grouped and this phase is called shuffling

>>> lines = sc.textFile("/public/randomtextwriter/part-m-00000")

>>> words = lines.flatMap(lambda line: line.split(" "))

>>> wordTuples = words.map(lambda word: (word,1))

>>> wordTupleGBK = wordTuples.groupByKey()

>>> wc = wordTupleGBK.map(lambda t: (t[0], sum(t[1])))

>>> wc.count()

1588631 shuffle read ansd shuffill write ---🡪61.2 MB of data

AggregateByKey :

**agregateByKey** function aggregates the values of each key with using given functions (seqFunc, combFunc) and a neutral zero value. seqFunc runs firstly and it makes the given calculation with zeroValue and current value and then create an output value.

**aggregateByKey**() is almost identical to reduceByKey() (both calling combineByKey() behind the scenes)

>>> orderItems = sc.textFile("/user/abhilashkalva/pyspark/orderItemsPartitioned")

>>> orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.s plit(",")[4])))

>>> for i in orderItemsMap.take(3): print(i) ...

(5, 99.96)

(5, 299.98)

(5, 129.99)

>>> revenueAndCountPerOrder = orderItemsMap.reduceByKey(lambda x,y: (x[0]+y[0],x)

>>> for i in revenueAndCountPerOrder.take(5): print(i)

...

(32768, (1029.94, 4))

(49152, (299.98, 1))

(32772, (499.9, 2))

(25096, (969.9000000000001, 5))

(8, (729.8399999999999, 4))

aggregateByKey is very rarly used for aggregations and reduceByKey ic most commonly used.

>>> orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.s>>> revenueCountperOrder = 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])

... )

>>> for i in revenueCountperOrder.take(10): print(i)

...

(32768, (1029.94, 4))

(49152, (299.98, 1))

(32772, (499.9, 2))

(25096, (969.9000000000001, 5))

(8, (729.8399999999999, 4))

(60076, (399.98, 1))

(65468, (1079.93, 5))

(32780, (579.86, 4))

(16, (419.93, 2))

(38232, (599.95, 2))

When computing intermidiat values and final value use aggregateByKey

Apache Spark 2 - Basic Transformations and Actions 03 - filter, joins and sortByKey

Compute daily Revenue

>>> orders = sc.textFile("/public/retail\_db/orders")

>>> for i in orders.take(4): print(i)

...

1,2013-07-25 00:00:00.0,11599,CLOSED

2,2013-07-25 00:00:00.0,256,PENDING\_PAYMENT

3,2013-07-25 00:00:00.0,12111,COMPLETE

4,2013-07-25 00:00:00.0,8827,CLOSED

>>> orders.map(lambda s: s.split(",")[3])

PythonRDD[3] at RDD at PythonRDD.scala:48

>>> orders.map(lambda s: s.split(",")[3]).take(10)

['CLOSED', 'PENDING\_PAYMENT', 'COMPLETE', 'CLOSED', 'COMPLETE', 'COMPLETE', 'COMPLETE', 'PROCESSING', 'PENDING\_PAYMENT', 'PENDING\_PAYMENT']

>>> orders.map(lambda s: s.split(",")[3]).distinct().take(10) or .collect()

['CLOSED', 'CANCELED', 'COMPLETE', 'PENDING\_PAYMENT', 'SUSPECTED\_FRAUD', 'PENDING', 'ON\_HOLD', 'PROCESSING', 'PAYMENT\_REVIEW']

In this closed and complete are considered as revenue

Whatever logic we passed to the lambda function it has to be python logic only. We cannot use spark APIs as part of the lambda function

As part of lambda functions which are passed to APIs such as map, flatMap, filter etc – the logic should be pure python

# filter, join, aggregation, sorting

>>> ordersFiltered = orders.filter(lambda o: o.split(",")[3] in ('CLOSED', 'COMPLETE'))

>>> type(ordersFiltered)

<class 'pyspark.rdd.PipelinedRDD'>

>>> orders.count()

68883

>>> ordersFiltered.count()

30455

>>> for i in ordersFiltered.take(6): print(i)

...

1,2013-07-25 00:00:00.0,11599,CLOSED

3,2013-07-25 00:00:00.0,12111,COMPLETE

4,2013-07-25 00:00:00.0,8827,CLOSED

5,2013-07-25 00:00:00.0,11318,COMPLETE

6,2013-07-25 00:00:00.0,7130,COMPLETE

7,2013-07-25 00:00:00.0,4530,COMPLETE

>>> orderItems = sc.textFile("/public/retail\_db/order\_items")

>>> for i in orderItems.take(4): print(i)

...

1,1,957,1,299.98,299.98

2,2,1073,1,199.99,199.99

3,2,502,5,250.0,50.0

4,2,403,1,129.99,129.99

# 6,2013-07-25 00:00:00.0,7130,COMPLETE --.> (6, ‘2013-07-25 00:00:00.0’)

# 2,2,1073,1,199.99,199.99 -------------🡪 (2, 199.99)

>>> orderFilterdMap = ordersFiltered.map(lambda o: (int(o.split(",")[0]),o.split (",")[1]))

>>> orderItemsMap = orderItems.map(lambda oi: (int(oi.split(",")[1]), float(oi.split(",")[4])))

>>> for i in orderFilterdMap.take(5): print(i)

(1, '2013-07-25 00:00:00.0')

(3, '2013-07-25 00:00:00.0')

(4, '2013-07-25 00:00:00.0')

(5, '2013-07-25 00:00:00.0')

(6, '2013-07-25 00:00:00.0')

>>> for i in orderItemsMap.take(5):print(i)

(1, 299.98)

(2, 199.99)

(2, 250.0)

(2, 129.99)

(4, 49.98)

...

# (5, ‘2013-07-25 00:00:00.0’) (5, 299.98) 🡪 (5, (‘2013-07-25 00:00:00.0’, 299.98))

>>> ordersJoin = orderFilterdMap.join(orderItemsMap)

>>> for i in ordersJoin.take(5): print(i)

...

(34576, ('2014-02-23 00:00:00.0', 49.98))

(34576, ('2014-02-23 00:00:00.0', 299.98))

(34576, ('2014-02-23 00:00:00.0', 59.99))

(34576, ('2014-02-23 00:00:00.0', 124.95))

(34576, ('2014-02-23 00:00:00.0', 299.98))

>>> orders.map(lambda o: o.split(",")[0]).distinct().count()

68883

>>> orderItems.map(lambda oi: oi.split(",")[1]).distinct().count()

57431

>>> ordersJoin = orderFilterdMap.join(orderItemsMap)

>>> ordersJoinMap = ordersJoin.map(lambda o: o[1])

>>> for i in ordersJoinMap.take(4): print(i)

...

(u'2014-05-16 00:00:00.0', 299.98)

(u'2014-05-16 00:00:00.0', 149.94)

(u'2014-05-16 00:00:00.0', 299.98)

(u'2014-05-16 00:00:00.0', 149.94)

>>> ordersJoinMap.count()

75408

>>> dailyRevenue = ordersJoinMap.reduceByKey(lambda x,y: x+y) / from operator import add

/ dailyRevenue = ordersJoinMap.reduceByKey(add)

>>> for i in dailyRevenue.take(5): print(i)

...

(u'2013-09-19 00:00:00.0', 51600.33999999997)

(u'2013-11-29 00:00:00.0', 54290.369999999966)

(u'2013-12-06 00:00:00.0', 67137.96999999997)

(u'2014-06-17 00:00:00.0', 37943.72999999998)

(u'2013-11-07 00:00:00.0', 56618.05999999998)

>>> dailyRevenueSorted = dailyRevenue.sortByKey() dailyRevenue.sortByKey(False) descending

>>> for i in dailyRevenueSorted.take(10) : print(i)

...

(u'2013-07-25 00:00:00.0', 31547.229999999992)

(u'2013-07-26 00:00:00.0', 54713.22999999997)

(u'2013-07-27 00:00:00.0', 48411.47999999998)

(u'2013-07-28 00:00:00.0', 35672.029999999984)

(u'2013-07-29 00:00:00.0', 54579.69999999998)

(u'2013-07-30 00:00:00.0', 49329.289999999964)

(u'2013-07-31 00:00:00.0', 59212.48999999996)

(u'2013-08-01 00:00:00.0', 49160.07999999997)

(u'2013-08-02 00:00:00.0', 50688.57999999997)

(u'2013-08-03 00:00:00.0', 43416.739999999976)

>>> orderItems = sc.textFile("/public/retail\_db/order\_items")

>>> oiMap = orderItems.map(lambda oi: ((int(oi.split(",")[1]), float(oi.split(",")[4])),oi))

>>> for i in oiMap.take(3): print(i)

...

((1, 299.98), u'1,1,957,1,299.98,299.98')

((2, 199.99), u'2,2,1073,1,199.99,199.99')

((2, 250.0), u'3,2,502,5,250.0,50.0')

>>> for i in oiMap.sortByKey(True).take(10):print(i)

...

((1, 299.98), u'1,1,957,1,299.98,299.98')

((2, 129.99), u'4,2,403,1,129.99,129.99')

((2, 199.99), u'2,2,1073,1,199.99,199.99')

((2, 250.0), u'3,2,502,5,250.0,50.0')

((4, 49.98), u'5,4,897,2,49.98,24.99')

((4, 150.0), u'7,4,502,3,150.0,50.0')

((4, 199.92), u'8,4,1014,4,199.92,49.98')

((4, 299.95), u'6,4,365,5,299.95,59.99')

((5, 99.96), u'11,5,1014,2,99.96,49.98')

((5, 129.99), u'13,5,403,1,129.99,129.99')

>>> for i in oiMap.sortByKey(False).take(10):print(i)

...

((68883, 1999.99), u'172197,68883,208,1,1999.99,1999.99')

((68883, 150.0), u'172198,68883,502,3,150.0,50.0')

((68882, 59.99), u'172195,68882,365,1,59.99,59.99')

((68882, 50.0), u'172196,68882,502,1,50.0,50.0')

((68881, 129.99), u'172194,68881,403,1,129.99,129.99')

((68880, 250.0), u'172190,68880,502,5,250.0,50.0')

((68880, 249.9), u'172192,68880,1014,5,249.9,49.98')

((68880, 199.99), u'172191,68880,1073,1,199.99,199.99')

((68880, 149.94), u'172189,68880,1014,3,149.94,49.98')

((68880, 149.94), u'172193,68880,1014,3,149.94,49.98')

>>> for i in oiMap.sortByKey(True).map(lambda o: o[1]).take(10):print(i)

...

1,1,957,1,299.98,299.98

4,2,403,1,129.99,129.99

2,2,1073,1,199.99,199.99

3,2,502,5,250.0,50.0

5,4,897,2,49.98,24.99

7,4,502,3,150.0,50.0

8,4,1014,4,199.92,49.98

6,4,365,5,299.95,59.99

>>> for i in dailyRevenueSorted.map(lambda o: o[0] + "," + str(o[1])).take(3):

... print(i)

...

2013-07-25 00:00:00.0,11516.91

2013-07-26 00:00:00.0,12547.35

2013-07-27 00:00:00.0,15255.27

>> dailyRevenueSorted.map(lambda o: o[0]+","+str(o[1])).saveAsTextFile("/user/abhilashkalva/pyspark/daily\_revenue")

26-12-2020

Apache Spark 2 - Advanced Transformations such as ranking using groupByKey

mapPartitions:

>>> l = ["Hello World","how are you"]

>>> list(map(lambda l: l.split(" "),l))

[['Hello', 'World'], ['how', 'are', 'you']]

>>> from itertools import chain

>>> list(chain.from\_iterable(map(lambda l: l.split(" "),l)))

['Hello', 'World', 'how', 'are', 'you']

chain.from\_iterable -🡪 concatenate lists

>>> l = ["Hello World","how are you"]

>>> m = list(map(lambda l: l.split(" "),l))

>>> m

[['Hello', 'World'], ['how', 'are', 'you']]

>>> from itertools import chain

>>> w = chain.from\_iterable(m)

>>> len(list(w) )

5

>>> t = map(lambda w: (w,1),w)

>>> list(t)

[('Hello', 1), ('World', 1), ('how', 1), ('are', 1), ('you', 1)]

>>>

>>> def getWordTuples(i):

... import itertools as it

... wordTuples = map(lambda s: (s,1),it.chain.from\_iterable(map(lambda s: s.split(" "),i)))

... return worldTuples

[('Hello', 1), ('World', 1), ('how', 1), ('are', 1), ('you', 1)]

>>> 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

...

>>> lines = sc.textFile("/public/randomtextwriter/part-m-00000")

>>> wordTuples = lines.mapPartitions(lambda i: getWordTuples(i))

>>> for i in wordTuples.reduceByKey(lambda x,y:x+y).take(10): print(i)

...

('�\x03�frameable', 193)

('uninterpleaded', 97987)

('uloid', 97456)

('pyrocatechol', 96995)

('horsefly', 97843)

('sud', 97550)

('archesporial', 97701)

('decidable', 97366)

('codisjunct', 97280)

('\x00\x00\x02�\x00\x00\x0054Effie', 2)

Ranking using groupByKey:

1 – global ranking

2 – byKey ranking

>>> products = sc.textFile("/public/retail\_db/products")

>>> products.filter(lambda p: p.split(",")[4] == "").first()

'685,31,"TaylorMade SLDR Irons - (Steel) 4-PW, AW",,899.99,http://images.acmesports.sports/TaylorMade+SLDR+Irons+-+%28Steel%29+4-PW%2C+AW'

>>> productsFiltered = products.filter(lambda p: p.split(",")[4] != "")

>>> products.count()

1345

>>> productsFiltered.count()

1344

>> for i in productsFiltered.take(2): print(i)

...

1,2,Quest Q64 10 FT. x 10 FT. Slant Leg Instant U,,59.98,http://images.acmesports.sports/Quest+Q64+10+FT.+x+10+FT.+Slant+Leg+Instant+Up+Canopy

2,2,Under Armour Men's Highlight MC Football Clea,,129.99,http://images.acmesports.sports/Under+Armour+Men%27s+Highlight+MC+Football+Cleat

>>>

Top N products with in each category:using price as criteria

>>> productsFilteredMap = productsFiltered.map(lambda p: (int(p.split(",")[1]),p))

>>> for i in productsFilteredMap.take(1):print(i) ...

(2, '1,2,Quest Q64 10 FT. x 10 FT. Slant Leg Instant U,,59.98,http://images.acmesports.sports/Quest+Q64+10+FT.+x+10+FT.+Slant+Leg+Instant+Up+Canopy')

>>> productsMap = productsFiltered.map(lambda p: (int(p.split(",")[1]),p))

>>> productsGBKcategory = productsMap.groupByKey()

>>> for i in productsGBKcatogory.take(2): print(i)

...

(2, <pyspark.resultiterable.ResultIterable object at 0x6eb1f5117e80>)

(4, <pyspark.resultiterable.ResultIterable object at 0x6eb1f5117fd0>)

# >>> t = productsGBKcategory.first()

>>> t

(2, <pyspark.resultiterable.ResultIterable object at 0x6eb2060c9358>)

>>> l = list(t[1])

>>>sorted(l,key = lambda p: float(p.split(“,”)[4]), reverse=True)[:4]

>>>def getTopNProducts(products, topN):

Return sorted(products, key = lambda p: float(p.split(“,”)[4]), reverse=True)[:topN]

>>>getTopNProducts(l,4)

>>>productsGBCategory

>>>topNproductsByCategory = productsGBCategory.map(lambda p: getTopNProducts(list(p[1]),4))

>>>for i in topNProductsByCategory.take(10): print(i)

## >>>topNproductsByCategory = productsGBCategory.map(lambda p: getTopNProducts(list(p[1]),4))

>>>for i in topNProductsByCategory.take(10): print(i)

//

>>>List(map(lambda p: float(p.split(“,”)[4]),l))

>>>set(map(lambda p: float(p.split(“,”)[4]),l))

>>>sorted((set(map(lambda p: float(p.split(“,”)[4]),l)),reverse=True)

>>>topNPrices = sorted((set(map(lambda p: float(p.split(“,”)[4]),l)),reverse=True)

>>>lSorted = sorted(l, key=float(p.split(“,”)[4]),reverse=True)

>>>for i in lSorted: print(i)

For i in list(it.takewhile(lambda p: float(p.split(“,”)[4]) in topNPrices,lSorted)):

print(i)

# we got all 6 records which fall under Top 4 priced products

---------------------------------------------------------------------------------------------------

>>> def getTopNPricedProducts(products, topN):

... import itertools as it

... productPrices = sorted(set(map(lambda p: float(p.split(",")[4]),products)) ,reverse=True)[:topN]

... productsSorted = sorted(products, key=lambda p: float(p.split(",")[4],reve rse =True)

... return it.takewhile(lambda p: float(p.split(",")[4]) in productPrices,prod uctsSorted)

File "<stdin>", line 5

return it.takewhile(lambda p: float(p.split(",")[4]) in productPrices,produc tsSorted)

>>>topNPricedProductsByCategory=productsGBCategory.flatMap(lambda p: getTopNPricedProducts(list(p[1]),4))

>>>for i in topNPricedProductsByCategory.take(10):print(i)

-----

------

topNPricedProductsByCategory.count()

432

Map ,filter,functools.reduce, itertools.takewhile, itertools.chain, pandas

30-12-2020

Apache Spark 2 - Development and Deployment Life Cycle

Daily\_revenue:

from pyspark import SparkConf, SparkContext  
import configparser as cp  
import sys  
props = cp.RawConfigParser()  
props.read("src/main/resources/application.properties")  
env = sys.argv[1]  
  
conf = SparkConf().setMaster(props.get(env,'executionMode')).setAppName("Daily Revenue").set("conf.ui.port", "12901")  
sc = SparkContext(conf = conf)  
orders = sc.textFile(props.get(env, 'input.base.dir')+"/orders")  
order\_items = sc.textFile(props.get(env, 'input.base.dir')+"/order\_items")  
ordersFiltered = orders.filter(lambda o: o.split(",")[3] in ('CLOSED', 'COMPLETE'))  
orderFilterdMap = ordersFiltered.map(lambda o: (int(o.split(",")[0]),o.split (",")[1]))  
orderItemsMap = order\_items.map(lambda oi: (int(oi.split(",")[1]), float(oi.split(",")[4])))  
ordersJoin = orderFilterdMap.join(orderItemsMap)  
ordersJoinMap = ordersJoin.map(lambda o: o[1])  
dailyRevenue = ordersJoinMap.reduceByKey(lambda x,y: x+y)  
dailyRevenueSorted = dailyRevenue.sortByKey()  
dailyRevenueSortedMap = dailyRevenueSorted.map(lambda o: o[0]+","+str(o[1]))  
dailyRevenueSortedMap.saveAsTextFile(props.get(env, 'output.base.dir')+"/daily\_revenue\_app")

then code is push to the cluster

cd projectpath

C:/../pycharmprojects> scp -r bootcamppyspark weburl:~

run in cluster

[--- bootcampPyspark]$ spark-submit --master yarn \

--deploy-mode client \

--conf spark.ui.port=12901 \

--conf spark.dynamicAllocation.enabled=false \

Src/mail/python/retail\_db/DailyRevenue.py prod

[--- bootcampPyspark]$ hadoop fs -ls /user/training/bootcamp/pyspark/daily\_Revenue\_app01

30-12-2020

Apache Spark 2 - Accumulators, Broadcast Variables, Repartition and Coalesce - 01

Prob: compute revenue per product for a given month

Sc ----🡪is a driver program when we submit the spark jobs Is nothing but a JVM

The JVM will have expose to all the APIs that are required to run the spark jobs successfully

\_gateway --🡪access all those APIs that are part of the JVM that will be used to run our Spark Jobs

import sys  
import configparser as cp  
  
try:  
 from pyspark import SparkContext,SparkConf  
  
 props = cp.RawConfigParser()  
 props.read("src/main/resources/application.properties")  
  
 conf = SparkConf().setAppName("Total Revenue Per Day").setMaster("local")  
  
 sc =SparkContext(conf = conf)  
 # inputPath = sys.argv[1]  
 # outputPath = sys.argv[2]  
 inputPath = "C:\\Users\\ABHILASH KALVA\\Desktop\\data engineer\\itversity\\data-master\\retail\_db"  
 outputPath = "C:\\Users\\ABHILASH KALVA\\Desktop\\data engineer\\spark\\pyspark\\daily\_revenue"  
 month = sys.argv[1]  
  
  
 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))):  
 print("Input Path Does NOt Exists")  
 #else:  
 #if(fs.exists(Path(outputPath))):  
 # fs.delete(Path(outputPath),True)  
  
 orders = inputPath + "\\orders\\orders"  
 ordersFiltered = sc.textFile(orders). \  
 filter(lambda order: month in order.split(",")[1]). \  
 map(lambda order:(int(order.split(",")[0]),1))  
 orderItems = inputPath + "\\order\_items\\orderItems"  
 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)  
  
 localPath = sys.argv[4]  
 productFile = open(localPath + "/products/products")  
 products = productFile.read().splitlines()  
  
 sc.parallelize(products). \  
 map(lambda product:  
 (int(products.split(",")[0]),products.split(",")[2])  
 ). \  
 join(revenueByProductId). \  
 map(lambda product: product[1][0] + "\t" + str(product[1][1])). \  
 saveASTExtFile(outputPath)  
  
except ImportError as e:  
 print("Can Not Import Spark Modules",e)  
sys.exit(1)

Accumulator is a kind of variable shared across all the workers which can be used to write or update some value. This is a write only variable for example v1 is my accumulator and workers are used to update the value by one are increment the value by one based on the blank line or we can write any logic instead of blank line our own custom logic based on which we want to update this variable.

Here driver is creating the accumulator. Updated state available with in driver program. this can help you to maintain counters.

**Accumulators** are variables that are only “added” to through an associative operation and can therefore, be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. **Spark** natively supports **accumulators** of numeric types, and programmers can add support for new types.

Accumulators is a shared variable which can be used to implement counters with in the Spark Application

(Only purpose of the accumulators is store in to the databases we can see how the data is loaded inner basis for all those important metrics we can store into a data base and we can actually trouble soot the issues or we can actually setup alert mechanism incase any obnormal number we see with this accumulator.)

It is important to perform some counts as part of application for

* Unit testing
* Data quality

These counters cannot be global variables as part of the program

Instead we need to use accumulator which will be managed by spark

Accumulators will be passed to executors and scope is managed across all the executors or executor tasks

Accumulators can be used to any spark APIs

sc.accumulator() is the api to create accumulator

in any Spark API,we can increment the accumulator

As python lambda functions directly cannot increment accumulator we will create a function which will be invoked as the part of the spark API such as map,filter

🡪Take revenue per product for given month program and add below accumulators

* To get number of orders for the month – ordersCount
* To get number of orderItems for the month – orderItemsCount
* Increment ordersCount as part of map function ofter filtering on the month
* Increment orderItemsCount as part of map function after join

Broadcast variable is a shared variable which are used for reading purpose.

sc.broadcast(codeMap) -🡪 here we broadcasted the variable the shared it is available to all the worker nodes but the worker nodes are using this variables to read some data. Ths can be use as a caching mechanism broadcasting some variable with in spark job

**Broadcast variables** in Apache **Spark** is a mechanism for sharing **variables** across executors that are meant to be read-only. Without **broadcast variables** these **variables** would be shipped to each executor for every transformation and action, and this can cause network overhead.

Broadcast variable is another type of shared variable which can be broadcasted into all the executors and can access at run time by tasks while processing data. It is typically used to replace joins with lookups when very large dataset is joined with small data set which can fit into 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 preliminary type or it could be a hash map
* Here are 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 small data set(Dimention),broadcasting dimention can have considerable performance improvement.
* Broadcast variables are immutable
* We can read data from HDFS or local file system or even as configuration parameters
* Broadcast using broadcast method of Spark Context

The **repartition** method can be used to either increase or decrease the number of partitions in a DataFrame. ... The **repartition** algorithm does a full data shuffle and equally distributes the data among the partitions. It does not attempt to minimize data movement like the coalesce algorithm.

The **coalesce** method reduces the number of partitions in a DataFrame. **Coalesce** avoids full shuffle, instead of creating new partitions, it shuffles the data using Hash Partitioner (Default), and adjusts into existing partitions, this means it can only decrease the number of partitions.

Apache Spark 2 - Accumulators, Broadcast Variables, Repartition and Coalesce – 02

Repartition Vs Coalesce :

both are the transformation api(s) provided by the apache spark for changing the underlying partitioning of the dataset.

* Repartition can be used to increase and decrease the number of partition on the datasets.
* Coalesce can only be used to only decrease the number of partition on the datasets.
* Coalesce uses existing partitions to minimize the amount of data shuffling.
* Coalesce avoid the full shuffing.

EXAMPLE:

Lets say our data has four partitions-

Partition A 🡪 1,2,3

Partition B 🡪 4,5

Partition C 🡪 6,7,8

Partition D 🡪 9,10

Repartition :

newDF = df.repartition(2)

Part A 🡪 1,2,3,4,5,6

Part B 🡪 7,8,9,10

Here , the partition data lis moved from B,C 🡪 A and C 🡪 D, so shuffling is less. The algo is repartition went for full shuffle to evenly distribute the data.

* Coalesce may run faster than repartition.
* However Coalesce results in uneven, unequal sized partition and spark will generally work slower unequal sized partitions.
* Repartition does full shuffile to make sure that the partitions are equally distributed.
* That’s the reason ,Repartitioning will always results in equal sized partitions.
* Spark works faster with equal sized partitions

01-01-2021

DATA FRAMES

* A distributed collection of structured data.
* Similar to database tables
* Have a Schema.

SparkSession is the entry point to programming Spark with the Dataset and DataFrame API.

df = spark.read.options(header="true",inferSchema="true",nullValue="NA",timestampFormat="yyyy-MM-dd'T'HH:mm:ss",mode="failfast").csv("/user/abhilashkalva/abhi/survey.csv")

>>> type(df)

<class 'pyspark.sql.dataframe.DataFrame'>

Data Frame is nothing but RDD with structure and it provides ApIs which are lot more intuitive than primitive operations on top of RDD.

* Creating data frame 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, avro etc
* Reading text data with custom delimiters
* Compression concepts and algorithms

Creating data frame and pre defined functions.

--🡪 how to create data frames from different data sources(one data source can be a text file, another data source can be Hive , another data source can be relational databases using JDBC)

[abhilashkalva@gw02 ~]$ export PYSPARK\_PYTHON=python3

[abhilashkalva@gw02 ~]$ export SPARK\_MAJOR\_VERSION=2

[abhilashkalva@gw02 ~]$ cat ~/.bash\_profile

* Data Frame can be created on any data set which have structured associate with it.
* Attributes/columns in a data frame can be reffered using names.
* One can create 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 data frames
* show – to preview data(default 20 records)
* describe – to understand characteristics of data
* to get number of records
* collect – to converts data frame into array
* once data frame is created we can process data using two 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 object of type Spark Session
* When we launch pyspark it will automatically create a object of type sparkSession
* If we want manually create(if we want to use our IDE) follow this approach

from pyspark.sql import SparkSession

spark = SparkSession.builder.master(“local”).appName(“create dataframe over JDBC”).getOrCreate()

>>> ordersDF = spark.read.csv('/public/retail\_db/orders')

>>> type(ordersDF)

<class 'pyspark.sql.dataframe.DataFrame'>

>>> ordersDF.printSchema()

root

|-- \_c0: string (nullable = true)

|-- \_c1: string (nullable = true)

|-- \_c2: string (nullable = true)

|-- \_c3: string (nullable = true)

Each record have 4 fields which have given some generic names(\_C0,\_c1.....)

RDD is a lowest level data structure in spark. their is no schema associated with it. Where as data frame is a higher level data structure on top of rdd where the structure associated with the data

>>> ordersDF = spark.read.csv('/public/retail\_db/orders').toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status')

>>> ordersDF.printSchema()

root

|-- order\_id: string (nullable = true)

|-- order\_date: string (nullable = true)

|-- order\_customer\_id: string (nullable = true)

|-- order\_status: string (nullable = true)

>>> ordersDF.select('order\_id','order\_date')

DataFrame[order\_id: string, order\_date: string]

>>> ordersDF.select('order\_id','order\_date').show()

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

|order\_id| order\_date|

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

| 1|2013-07-25 00:00:...|

| 2|2013-07-25 00:00:...|

| 3|2013-07-25 00:00:...|

| 4|2013-07-25 00:00:...|

| 5|2013-07-25 00:00:...|

| 6|2013-07-25 00:00:...|

| 7|2013-07-25 00:00:...|

>>> for i in ordersDF.select('order\_id','order\_date').take(2):print(i)

...

Row(order\_id='1', order\_date='2013-07-25 00:00:00.0')

Row(order\_id='2', order\_date='2013-07-25 00:00:00.0')

Take(),collect() it will printing like row objects

>>> ordersDF.describe()

DataFrame[summary: string, order\_id: string, order\_date: string, order\_customer\_id: string, order\_status: string]

>>> ordersDF.describe().show() 🡪 some metrics about data

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

|summary| order\_id| order\_date|order\_customer\_id| order\_status|

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

| count| 68883| 68883| 68883| 68883|

| mean| 34442.0| null|6216.571098819738| null|

| stddev|19884.953633337947| null|3586.205241263963| null|

| min| 1|2013-07-25 00:00:...| 1| CANCELED|

| max| 9999|2014-07-24 00:00:...| 9999|SUSPECTED\_FRAUD|

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

>>> ordersDF.createTempView('orders')

>>> spark.sql('select \* from orders').show()

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

|order\_id| order\_date|order\_customer\_id| order\_status|

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

| 1|2013-07-25 00:00:...| 11599| CLOSED|

| 2|2013-07-25 00:00:...| 256|PENDING\_PAYMENT|

| 3|2013-07-25 00:00:...| 12111| COMPLETE|

| 4|2013-07-25 00:00:...| 8827| CLOSED|

First write createTempView then given name it and write sql like quiries.

Reading data from text files :-

from pyspark.sql import SparkSession  
spark = SparkSession.\  
 builder.\  
 master("local").\  
 appName("Getting Started").\  
 getOrCreate()  
  
ordersCSV = spark.\  
 read.\  
 csv('C:/Users/ABHILASH KALVA/Desktop/data engineer/retail\_db/orders/orders').\  
 toDF('order\_id', 'order\_date', 'order\_customer\_id', 'order\_status')  
  
# converting 'order\_id','order\_customer\_id' is string to integer  
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( )  
  
orderItemsCSV = spark.\  
 read.\  
 csv('C:/Users/ABHILASH KALVA/Desktop/data engineer/retail\_db/order\_items/order\_items').\  
 toDF('order\_item\_id', 'order\_item\_order\_id', 'order\_item\_product\_id',  
 'order\_item\_quantity','order\_item\_subtotal','order\_item\_product\_price')  
orderItemsCSV.printSchema( )  
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( )

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 data base name to table name while reading hive tables into DataFrame
* We can also run Hive Quiries directly using spark.sql
* Both spark.read.table and spark.sql returns Data Frame

>>> orders = spark.read.table('abhi\_retail\_db.orders')

>>> orders.printSchema()

root

|-- order\_id: integer (nullable = true)

|-- order\_date: string (nullable = true)

|-- order\_customer\_id: integer (nullable = true)

|-- order\_status: string (nullable = true)

>>> orders.show()

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

|order\_id| order\_date|order\_customer\_id| order\_status|

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

| 1|2013-07-25 00:00:...| 11599| CLOSED|

| 2|2013-07-25 00:00:...| 256|PENDING\_PAYMENT|

| 3|2013-07-25 00:00:...| 12111| COMPLETE|

Another approach is

orders = spark.sql("select \*from abhi\_retail\_db.orders")

Reading data from Relational database(here MySQL) over JDBC:-

[abhilashkalva@gw02 ~]$ pyspark --master yarn --conf spark.ui.port=12906 \

> --jars /usr/share/java/mysql-connector-java.jar \

> --driver-class-path /usr/share/java/mysql-connector-java.jar

from pyspark.sql import SparkSession  
spark = SparkSession.\  
 builder.\  
 master("local").\  
 appName("Getting Started").\  
 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.printSchema()  
orders.show()

or

from pyspark.sql import SparkSession  
spark = SparkSession.\  
 builder.\  
 master("local").\  
 appName("Getting Started").\  
 getOrCreate()  
  
orders = spark.read. \  
 jdbc('jdbc:mysql://ms.itversity.com','retail\_db.orders',  
 numPartitions=4,  
 properties={'user':'retail\_user','password':'itversity'})  
orders.printSchema()  
orders.show()

in this case we are reading data from remote data base into a dataframe

# down load mysql-connector-java-5.1.41.jar

# paste it to the spark-hadoop 2.7/jar location

from pyspark.sql import SparkSession  
spark = SparkSession.\  
 builder.\  
 master("local").\  
 appName("Getting Started").\  
 getOrCreate()  
  
  
orderItems = spark.read. \  
 jdbc('jdbc:mysql://ms.itversity.com','retail\_db.order\_items',  
 numPartitions=4,  
 column='order\_item\_order\_id',  
 lowerBound='10000',  
 upperBound='20000',  
 properties={'user':'retail\_user','password':'itversity'})  
orderItems.write.csv('C:/Users/ABHILASH KALVA/Desktop/data engineer/research/data/order\_items')

CreateProcess error=216, This version of %1 is not compatible with the version of Windows you're running. Check your computer's system information and then contact the software publisher

from pyspark.sql import SparkSession  
spark = SparkSession.\  
 builder.\  
 master("local").\  
 appName("Getting Started").\  
 getOrCreate()  
  
  
orderItems = spark.read. \  
 jdbc('jdbc:mysql://ms.itversity.com',  
 '(select order\_item\_order\_id, sum(order\_item\_subtotal) order\_revenue '  
 'from retail\_db.order\_items group by order\_item\_order\_id) q',  
 properties={'user':'retail\_user','password':'itversity'})  
orderItems.show()

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

|order\_item\_order\_id| order\_revenue|

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

| 1| 299.9800109863281|

| 2| 579.9800109863281|

| 4| 699.8500099182129|

| 5|1129.8600387573242|

Data Frame Operations –Overview

* Selection or Projection - select
* Filtering data – filter or where
* Aggregations – groupBy and agg with support of functions such as sum,avg,min,max etc
* Sorting – sort or orderBy
* Analytics Functions – aggregations,ranking and windowing functions
* Projecting data using select, withCoulmn and selectExpr

From pyspark.sql import functions

help(functions)

some functions in action:

string:

>>> orders.select(substring('order\_date', 1,7).alias('order\_month')).show()

2013-07

2013-07

Case:

>>> orders.select(lower(orders.order\_status).alias("order\_status")).show()

Only it will print order\_status column only

>> orders.withColumn("order\_status",lower(orders.order\_status)).show()

It will print all columns in orders table

trim:

date \_format:

>>> orders.withColumn('order\_month',date\_format(orders.order\_date’, ‘yyyyMM’)).show()

trunk: we want all the data from 1st of the month to till date

case when: categirize data into fewer groups

# >>> orders.selectExpr(‘case when order\_satatus in(“COMPLETE”,”CLOSED”) then “COMPLETED when order\_status ==”CANCELED” then “CANCELED” else “PENDING” end’)

03-01-2021

Apache Spark 2 - Data Frame Operations - Basic Transformations such as filtering, aggregations etc

>>> orders = spark.read.csv("/user/abhilashkalva/data/retail\_db/orders").toDF("order\_id","order\_date","order\_customer\_id","order\_status")

>>> orderItemsCSV = spark.read.csv("/user/abhilashkalva/data/retail\_db/order\_items.txt").toDF("order\_item\_id","order\_item\_order\_id","order\_item\_product\_id","order\_item\_quantity","order\_item\_subtotal","order\_item\_product\_price")

>>> orders.show()

Basic Transformations : Filter,join,groupBy,union,sort

>>> orders.select("order\_id","order\_status").show()

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

|order\_id| order\_status|

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

| 1| CLOSED|

| 2|PENDING\_PAYMENT|

| 3| COMPLETE|

Or

>>> orders.select(orders.order\_id,orders.order\_status).show()

>>> from pyspark.sql import functions as f

>>> orders.select(orders.order\_id,orders.order\_date,f.date\_format(orders.order\_date, "YYYYMM")).show()

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

|order\_id| order\_date|date\_format(order\_date, YYYYMM)|

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

| 1|2013-07-25 00:00:...| 201307|

| 2|2013-07-25 00:00:...| 201307|

Or

>>> from pyspark.sql.functions import date\_format

>>> orders.select(orders.order\_id,orders.order\_date,date\_format(orders.order\_date, "YYYYMM")).show()

Or date\_format(orders.order\_date, "YYYYMM").alias("order\_month")

>>> from pyspark.sql.functions import \* (imported All functions)

>>> orders.select('\*').show()

* withColumn --- diff alias name we get us additional field. Same alias name get a same field

>>> orders.withColumn('order\_month',date\_format(orders.order\_date, 'YYYYMM')).show()

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

|order\_id| order\_date|order\_customer\_id| order\_status|order\_month|

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

| 1|2013-07-25 00:00:...| 11599| CLOSED| 201307|

* selectExpr:-

>>> orders.selectExpr('date\_format(order\_date,"YYYYMM") as date\_month').show()

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

|date\_month|

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

| 201307|

| 201307|

Data Frame have two APIs to filter data, where and filter

Where:-

Traditiona SQL style

>>> orders.where('order\_status = "COMPLETE"').show() or >>> orders.where(orders.order\_status == "COMPLETE").show()

>>> orders.where('order\_status in("COMPLETE","CLOSED")').show()

* prob: get orders which are either complete or closed and placed in month of 2013 aug

>>> orders.where(orders.order\_status.isin('COMPLETE','CLOSED').\_\_and\_\_(date\_format(orders.order\_date,'YYYYMM')=='201308')).show()

Using like

orders.where("order\_status in ('COMPLETE','CLOSED') and order\_date like '2013-08%'").show()

using date\_format()

>>> orders.where("order\_status in ('COMPLETE','CLOSED') and date\_format(order\_date,'YYYYMM') = '201308'").show()

Data frame style

>>> orders.where((orders.order\_status == 'COMPLETE').\_\_or\_\_(orders.order\_status == 'CLOSED')).show()

>>> orders.where(orders.order\_status.isin('COMPLETE','CLOSED')).show()

Filter:-

>>> orders.filter("order\_id > 450").show()

* 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()

* get all the orders which are placed on first of every month

>>> orders.where("date\_format(order\_date,'dd') = '01'").show()

Data frame style

>>> from pyspark.sql.functions import date\_format

>>> orders.where(date\_format(orders.order\_date,'dd')== '01').show()

Joining datasets :

* ## get daily product revenue

orders - ("order\_id","order\_date","order\_customer\_id","order\_status")

orderItems - ("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

>>> orders.where("order\_status in('COMPLETE','CLOSED')"). \

... join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id).show()

>>> orders.join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id, 'left').show()

**>>> orders.join(orderItems, orders.order\_id == orderItems.order\_item\_order\_id, 'left'). \**

**... where(orderItems.order\_item\_order\_id.isNull()).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()**

Aggregations:-

>>> orderItems.filter("order\_item\_order\_id = 2").agg(sum("order\_item\_subtotal")).show()

Compute the revenue each order\_id:

>>> orderItems.groupBy('order\_item\_order\_id').sum('order\_item\_subtotal').show()

>>> orderItems.groupBy('order\_item\_order\_id').agg(round(sum('order\_item\_subtotal'),2).alias('order\_revenue')).show()

Or

>>> orderItems.groupBy(orderItems.order\_item\_order\_id).agg(round(sum('order\_item\_subtotal'),2 ).alias('order\_revenue')).show()

>>> orders.groupBy('order\_status').count().show()

>>> from pyspark.sql.functions import \*

>>> orders.groupBy('order\_status').agg(count('order\_status').alias('status\_count')).show()

>>> 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('order\_item\_subtotal'),2).alias('revenue')

>>>dailyProductRevenue.show()

Sorting data:

Sort or orderBy can be used to sort the data

>>> orders.orderBy('order\_status').show()

Composit sorting

>>> orders.orderBy('order\_date','order\_status').show()

>>> orderItems.orderBy(orderItems.order\_item\_order\_id,orderItems.order\_item\_subtotal.desc()).show()

>>> dailyProductRevenue.orderBy('order\_date',dailyProductRevenue.revenue.desc()).show()

Saving file

>>> dailyProductRevenue.write.csv("/user/abhilashkalva/daily\_product\_revenue")

Apache Spark 2 - Data Frame Operations - Analytics Functions or Windowing Functions

>>> from pyspark.sql.window import Window

>>> help(Window)

Compute revenue for each order\_id

>>> revenuePerOrder = orderItems.groupBy('order\_item\_order\_id').agg(round(sum('order\_item\_subtotal'),2).alias(‘order\_revenue’))

>>>revenuePerOrder.show()

>>> ordereItems.join(revenuePerOrder, orderItems.order\_item\_order\_id == revenuePerOrder.order\_item\_order\_id).show()

>>> ordereItems.join(revenuePerOrder, orderItems.order\_item\_order\_id == revenuePerOrder.order\_item\_order\_id). \

... select(orderItems.order\_item\_order\_id,'order\_item\_subtotal','order\_revenue'). \

... show()

>>> spec = Window.partitionBy(orderItems.order\_item\_order\_id)

>>> type(spec)

<class 'pyspark.sql.window.WindowSpec'>

>>> orderItems.select('order\_item\_order\_id','order\_item\_subtotal',sum(orderItems.order\_item\_order\_id).over(spec).alias('order\_revenue')).show()

Percentage of orderItem subtotal in orderRevenue

>>> orderItems.select('order\_item\_order\_id','order\_item\_subtotal',round(orderItems.order\_item\_subtotal/sum(orderItems.order\_item\_order\_id).over(spec),2).alias('order\_revenue')).show()

>>> orderItems.withColumn('order\_revenue',sum('order\_item\_subtotal').over(spec)). \

... withColumn('order\_avg\_revenue', avg('order\_item\_subtotal').over(spec)). \

... show()

Using windowing functions:

* lead, lag
* we need to create WindowSpec object using partitionBy and then orderBy for most of the windowing functions
* some realistic use cases
* Salary difference b/w current and next/previous employee within each department

>>> from pyspark.sql.window import Window

>>> spec = Window.partitionBy('order\_item\_order\_id').orderBy(orderItems.order\_item\_subtotal.desc())

>>> orderItems.withColumn('order\_item\_revenue',

... lead('order\_item\_subtotal').over(spec)).show()

Apache Spark 2 - Spark SQL – Basic Transformations such as filtering, aggregations, joins etc

>>> ordersCSV = spark.read.csv("/user/abhilashkalva/data/retail\_db/orders").toDF("order\_id","order\_date","order\_customer\_id","order\_status")

>>> orderItemsCSV = spark.read.csv("/user/abhilashkalva/data/retail\_db/order\_items.txt").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( )

root

|-- order\_id: integer (nullable = true)

|-- order\_date: string (nullable = true)

|-- order\_customer\_id: integer (nullable = true)

|-- order\_status: string (nullable = true)

>>> 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( )

root

|-- order\_item\_id: integer (nullable = true)

|-- order\_item\_order\_id: integer (nullable = true)

|-- order\_item\_product\_id: integer (nullable = true)

|-- order\_item\_quantity: integer (nullable = true)

|-- order\_item\_subtotal: float (nullable = true)

|-- order\_item\_product\_price: float (nullable = true)

Writing sql like queries instead of data frame APIs to process data first we need to create temp view or in a memory view against this data frame

>>> orders.createOrReplaceTempView('orders')

>>> orderItems.createOrReplaceTempView('order\_items')

>>> spark.sql("set spark.hadoop.hive.metastore.warehouse.dir = /user/abhilashkalva/warehouse")

DataFrame[key: string, value: string]

>>> spark.sql('use abhi\_retail\_db')

DataFrame[]

>>> spark.sql('show tables').show()

>>> spark.sql('select \* from orders').show()

Relationship with 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.
* Tables 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 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 function etc
* Standard Hive commands(except SQL)
* Spark is type of SparkSession
* List of tables – spark.sql(“show tables”).show()
* Switch database – spark.sql(“use abhi\_retail\_db”).show()
* Describe table – spark.sql(“describe table orders”).show()
* Show functions – 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 savaASTable, insertInto etc to directly write data frame into Hive table

>>> spark.sql('create table orders\_hive as select \* from orders').show()

>>> spark.sql('show tables').show()

>>> for i in spark.sql('describe formatted orders\_hive').collect():print(i)

Show list of functions

>>> for f in spark.sql("show functions").collect():print(f)

>>> for f in spark.sql("describe function substring").collect():print(f)

>>> spark.sql('select substring("Hello World",7,3)').show()

>>> spark.read.table('orders\_hive').show()

>>> spark.read.table('abhi\_retail\_db.orders\_hive').show()

>>> spark.sql('select date\_format(current\_date, "YYYYMM")').show()

Filtering data – where clause:

>>> spark.sql('use abhi\_retail\_db').show()

>>> spark.sql('select \* from orders where order\_status = "COMPLETE"').show()

* Get orders which are either COMPLETE or CLOSED and placed in month of 2013 August

>>> spark.sql('select \* from orders where date\_format(order\_date, "YYYY-MM") = "2013-08"').show()

>>> spark.sql('select \* from orders where (date\_format(order\_date, "YYYY-MM") = "2013-08") and order\_status in("COMPLETE","CLOSED")').show()

Or

>>> spark.sql('select \* from orders where order\_date like "2013-08%" and order\_status in("COMPLETE","CLOSED")').show()

* 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()

Joining datasets:

* 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)
* Few examples
* Get all order items corresponding to COMPLETE or CLOSED orders

>>> spark.sql('select \* from order\_items where order\_item\_order\_id in (select order\_id from orders where order\_status in ("COMPLETE","CLOSED"))').show()

---- if we want to see the results orders also we have to perform join ----

---- if we want to write the query in multiple lines use single cout(‘’’ like this)

>>> 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()

Or legacy style

>>> spark.sql('''select \* from orders o,order\_items oi

... where o.order\_id = oi.order\_item\_order\_id and o.order\_status in ("COMPLETE","CLOSED")''').show()

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''').show()

Only show null

>>> spark.sql('''select \* from orders o left outer join order\_items oi

... on o.order\_id = oi.order\_item\_order\_id

... where order\_item\_order\_id is null''').show()

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()

* >>> spark.sql('''select \* from orders''').count()
* >>> spark.sql('''select count(1) from orders''').show()
* Given order\_id

>>> spark.sql('''select sum(order\_item\_subtotal) from order\_items where order\_item\_order\_id =2 ''').show()

* Each order id

>>> spark.sql('''select order\_item\_order\_id,round(sum(order\_item\_subtotal),2) from order\_items group by(order\_item\_order\_id)''').show()

>>> spark.sql('''select order\_item\_order\_id,sum(order\_item\_subtotal) from order\_items group by(order\_item\_order\_id) having sum(order\_item\_subtotal)> 400''').show()

get count by status from orders

>>> spark.sql('''select order\_status,count(\*) from orders group by order\_status''').show()

Get daily product revenue(order\_date and order\_item\_product\_id are part of keys,order\_item subtotal is used for aggregation)

>>> DailyProductRevenue = spark.sql('''select o.order\_date,oi.order\_item\_product\_id,

... round(sum(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''').show()

Sorting data

Sort orders by status

>>> spark.sql("select \* from orders order by order\_status").show()

Sort order by date and then by status

>>> spark.sql("select \* from orders order by order\_date,order\_status").show()

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()

Take daily product revenue data and sort in ascending order by date and then descending order by revenue

- 08-01-2021

Apache Spark 2 - Spark SQL – Analytics Functions or Windowing Functions

Analytical functions:

Analytical functions typically categorized into three

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) wher 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

All these analatycal functions can only work as part of select clause

>>> inputBaseDir = "/public/retail\_db"

>>> ordersCSV = spark.read.csv("/user/abhilashkalva/data/retail\_db/orders").toDF("order\_id","order\_date","order\_customer\_id","order\_status")

>>> orderItemsCSV = spark.read.csv("/user/abhilashkalva/data/retail\_db/order\_items.txt").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()))

>>> 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.sql('select sum(order\_item\_subtotal) over (partition by order\_item\_order\_id) from order\_items').show()

>>> spark.sql('select order\_item\_order\_id,sum(order\_item\_subtotal) order\_revenue from order\_items group by(order\_item\_order\_id)').show()

We cannot apply non aggregate functions as part of select clause that is the limitation with group by

>>> order\_revenue = spark.sql('select order\_item\_order\_id,round(sum(order\_item\_subtotal),2) order\_revenue from order\_items group by(order\_item\_order\_id)')

>> spark.sql('select o.order\_item\_id,o.order\_item\_order\_id, '

... 'o.order\_item\_subtotal from,q.order\_revenue '

... 'order\_items o join (select order\_item\_order\_id, '

... 'round(sum(order\_item\_subtotal),2)order\_revenue '

... 'from order\_items group by order\_item\_order\_id) q '

... 'on o.order\_item\_order\_id = q.order\_item\_order\_id').show()

Or

>> spark.sql('select o.order\_item\_id,o.order\_item\_order\_id, '

... 'o.order\_item\_subtotal, '

... 'round(sum(o.order\_item\_subtotal) over (partition by o.order\_item\_order\_id) ,2) order\_revenue from '

... 'order\_items o').show()

Get the percentage

>> spark.sql('select o.order\_item\_id,o.order\_item\_order\_id, '

'round(o.order\_item\_subtotal/sum(o.order\_item\_subtotal) over (partition by o.order\_item\_order\_id ),2) \* 100 subtotal\_pct from '

'order\_items o').show()

Assign rank

With in each order\_item\_order\_id group data by order\_id and sort the data in descending order by order\_item subtotal

>> spark.sql('select o.order\_item\_id,o.order\_item\_order\_id, '

'o.order\_item\_subtotal, '

'rank() over (partition by order\_item\_order\_id order by order\_item\_subtotal desc) rnk from '

'order\_items o').show()

Diff b/w the current record and next record

Use windowing functions

>> spark.sql('select o.order\_item\_id,o.order\_item\_order\_id, '

'o.order\_item\_subtotal, '

'lead(order\_item\_subtotal) over (partition by order\_item\_order\_id order by order\_item\_subtotal desc) rnk from '

'order\_items o').show()

Fetch five records based on ranks

Execution flow

from -> where and/or join -> group by -> having -> select -> order by

>> spark.sql('select o.order\_item\_id,o.order\_item\_order\_id, '

'o.order\_item\_subtotal, '

'rank() over (partition by order\_item\_order\_id order by order\_item\_subtotal desc) rnk from '

'order\_items o order by order\_item\_order\_id, rnk desc').show()

* Analytics functions only used order by clause..these analytics functions cannot be used any other clause than select in spark sql or hive

Get top N products per day

>>> DailyProductRevenue = spark.sql('''select o.order\_date,oi.order\_item\_product\_id,round(sum(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''')

>>> spark.sql('select \* from (select d.\*, '

... 'rank() over (partition by order\_date order by revenue desc) rnk '

... 'from daily\_product\_revenue d) q '

... 'where rank <=5 '

... 'order by order\_date, revenue desc').show()

10-01-2021

Apache Spark 2 - Compression Concepts and algorithms

Csv format:

>>> orders.write.format('csv').option('codec', 'gzip').save('/user/abhilashkalva/bootcamppyspark/pyspark/orders\_csv\_compressed')

Json format:

>>> orders.write.option('compression', 'gzip').json('/user/abhilashkalva/bootcamppyspark/pyspark/orders\_json\_compressed')

Parquet format:

>>> orders.write.parquet('/user/abhilashkalva/bootcamppyspark/pyspark/orders\_parquet\_compressed')

By default data will compressed using snappy

Let us try to compress the parquet data with gzip

>>> spark.conf.set('spark.sql.parquet.compression.codec','gzip')

>>> orders.write.parquet('/user/abhilashkalva/bootcamppyspark/pyspark/orders\_parquet\_compressed1')

avro format:

spark doesn’t support avro format because we need to download jar files.

>>> orders.write.format('com.databricks.spark.avro').save('/user/abhilashkalva/bootcamppyspark/pyspark/orders\_avro\_compressed',mode='overwrite')

Read data from compressed file

>>> spark.read.format('parquet').load('/user/abhilashkalva/b00tcamppyspark/pyspark/orders\_parquet\_compressed1').show()

11-01-2021

Apache Spark 2 - Different file formats and custom delimiters

* 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
* avro (3rd party)
* processing text data with custom delimiters
* persisting or caching data frames

Overview of write APIs – dataframe.write

* 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 databeses
* we can also connect to any 3rd party database using relevant plugin and preserve data over there.

>>> orders.write.format('json').save('/user/abhilashkalva/bootcamppyspark/pyspark/orders\_json')

Or

>>> orders.write.json('/user/abhilashkalva/bootcamppyspark/pyspark/orders\_json',mode = "overwrite")

Writing the data into mysql database

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’)

Or

orders.write. \

jdbc(“jdbc:mysql://ms.itversity.com”, table , mode = ‘append’,

properties = {“user”: “retail\_user”,

“password”: “itversity”})

Writing the data into Hive :

>>> orders.write.saveAsTable('abhi\_retail\_db.orders\_hive',mode='overwrite')

Overview of read APIs – spark.read

>>>orders = orders.read.format(‘json’).load('/user/abhilashkalva/bootcamppyspark/pyspark/orders\_json')

Or

>>>orders = orders.read.json('/user/abhilashkalva/bootcamppyspark/pyspark/orders\_json')