**SHELL COMMAND**

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

get shell command and change the permissions to

-rw-rw-r-- ==>> drwxrwxr-x

chmod 764 **sqoop.sh** --->>only owner can excute

chmod 774 **sqoop.sh** --->>others can also excute

first view shell command

how to run = **./sqoop.sh**

=========================================================

**LOADING DATA**

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

val dataRDD = sc.textFile("/user/cloudera/sqoop\_import/departments")

dataRDD**.collect().foreach(println)**

dataRDD.count()

===========================================================

**SAVE**

**-----------------------------------------------------------------------------**

dataRDD**.saveAsTextFile**("/user/cloudera/scalaspark/departments")

dataRDD**.saveAsSequenceFile**("/user/cloudera/scalaspark/departmentsSeq")

===========================================================

**WRITING DATA INTO JSON FORMAT**

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

departmentsData**.toJSON.saveAsTextFile**("/user/cloudera/scalaspark/departmentsJson")

===========================================================

**WORD COUNT**

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

val data = sc.textFile("/user/cloudera/wordcount.txt")

val dataFlatMap = data.**flatMap(x => x.split(" "))**

val dataMap = dataFlatMap**.map(x => (x, 1))**

val dataReduceByKey = dataMap**.reduceByKey((x,y) => x + y)**

dataReduceByKey**.saveAsTextFile**("/user/cloudera/wordcountoutput")

===========================================================

**JOINING**

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

val ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

val orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

**Initally same for both Pyspark and python they starts with 0**

val ordersParsedRDD = ordersRDD**.map(rec => (rec.split(",")(0).toInt, rec))**

val orderItemsParsedRDD = orderItemsRDD.map(rec => (rec.split(",")(1).toInt, rec))

val ordersJoinOrderItems = orderItemsParsedRDD**.join**(ordersParsedRDD)

val revenuePerOrderPerDay = ordersJoinOrderItems.map(t => (t.\_2.\_2.split(",")(1), t.\_2.\_1.split(",")(4).toFloat))

**1 2(combination of green and blue)**

**1 2**

**0 0 1 2 3 4 4 0 1 2 3**

(**2828**,(**7097,2828,403,1,129.99,129.99**,2828,2013-08-10 00:00:00.0,4952,SUSPECTED\_FRAUD))

**ORDER COUNT PER DAY**

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

val ordersPerDay = ordersJoinOrderItems.map(rec => rec.\_2.\_2.split(",")(1) + "," + rec.\_1).distinct()

\*\*\*\*\* very importnet

val ordersPerDayParsedRDD = ordersPerDay.map(rec => (rec.split(",")(0), 1))

val totalOrdersPerDay = ordersPerDayParsedRDD.reduceByKey((x, y) => x + y)

==========================================================

**REVENUE PER DAY FROM JOINED DATA**

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

val totalRevenuePerDay = revenuePerOrderPerDay.reduceByKey((total1, total2) => total1 + total2 )

totalRevenuePerDay.sortByKey().collect().foreach(println)

==========================================================

**JOINING ORDER COUNT PER DAY AND REVENUE PER DAY**

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

val finalJoinRDD = totalOrdersPerDay.join(totalRevenuePerDay)

finalJoinRDD.collect().foreach(println)

==========================================================

**MAX PRICKED PRODUCT FROM PRODUCTS TABLE**

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

val productsRDD = sc.textFile("/user/cloudera/sqoop\_import/products")

val productsMap = productsRDD.map(rec => rec)

productsMap.reduce((rec1, rec2) => (

if(rec1.split(",")(4).toFloat >= rec2.split(",")(4).toFloat)

rec1

else

rec2)

)

**AVERAGE**

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

val revenue = sc.textFile("/user/cloudera/sqoop\_import/order\_items").

map(rec => rec.split(",")(4).toDouble).

reduce((rev1, rev2) => rev1 + rev2)

val totalOrders = sc.textFile("/user/cloudera/sqoop\_import/order\_items").

map(rec => rec.split(",")(1).toInt).

distinct().

count()

===========================================================

**NUMBER OF ORDERS BY STATUS**

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

val ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

val ordersMap = ordersRDD.map(rec => (rec.split(",")(3), 1))

ordersMap.countByKey().foreach(println)

**#groupByKey is not very efficient for aggregations. It does not use combiner**

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

val ordersByStatus = ordersMap.groupByKey().map(t => (t.\_1, t.\_2.sum))

**#reduceByKey uses combiner - both reducer logic and combiner logic are same**

val ordersByStatus = ordersMap.reduceByKey((acc, value) => acc + value)

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

**#combineByKey can be used when reduce logic and combine logic are different**

val ordersByStatus = ordersMap.combineByKey(value => 1, (acc: Int, value: Int) => acc+value, (acc: Int, value: Int) => acc+value)

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

**#Both reduceByKey and combineByKey expects type of input data and output data are same**

**aggregateByKey can be used when reduce logic and combine logic is different**

**#Also type of input data and output data need not be same**

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

val ordersMap = ordersRDD.map(rec => (rec.split(",")(3), rec))

val ordersByStatus = ordersMap.aggregateByKey(0, (acc, value) => acc+1, (acc, value) => acc+value)

ordersByStatus.collect().foreach(println)

===========================================================

**NUMBER OF ORDERS BY ORDER STATE AND ORDER DATE**

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

**#Key orderDate and orderStatus**

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

val ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

val ordersMapRDD = ordersRDD.map(rec => ((rec.split(",")(1), rec.split(",")(3)), 1))

val ordersByStatusPerDay = ordersMapRDD.reduceByKey((v1, v2) => v1+v2)

ordersByStatusPerDay.collect().foreach(println)

===========================================================

**#TOTAL REVENUE PER DAY**

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

val ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

val orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

val ordersParsedRDD = ordersRDD.map(rec => (rec.split(",")(0), rec))

val orderItemsParsedRDD = orderItemsRDD.map(rec => (rec.split(",")(1), rec))

val ordersJoinOrderItems = orderItemsParsedRDD.join(ordersParsedRDD)

val ordersJoinOrderItemsMap = ordersJoinOrderItems.map(t => (t.\_2.\_2.split(",")(1), t.\_2.\_1.split(",")(4).toFloat))

val revenuePerDay = ordersJoinOrderItemsMap.reduceByKey((acc, value) => acc + value)

revenuePerDay.collect().foreach(println)

==========================================================

**AVERAGE REVENUE PER DAY**

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

val ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

val orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

val ordersParsedRDD = ordersRDD.map(rec => (rec.split(",")(0), rec))

val orderItemsParsedRDD = orderItemsRDD.map(rec => (rec.split(",")(1), rec))

val ordersJoinOrderItems = orderItemsParsedRDD.join(ordersParsedRDD)

val ordersJoinOrderItemsMap = ordersJoinOrderItems.map(t => ((t.\_2.\_2.split(",")(1), t.\_1), t.\_2.\_1.split(",")(4).toFloat))

val revenuePerDayPerOrder = ordersJoinOrderItemsMap.reduceByKey((acc, value) => acc + value)

val revenuePerDayPerOrderMap = revenuePerDayPerOrder.map(rec => (rec.\_1.\_1, rec.\_2))

val revenuePerDay = revenuePerDayPerOrderMap.aggregateByKey((0.0, 0))(

(acc, revenue) => (acc.\_1 + revenue, acc.\_2 + 1),

(total1, total2) => (total1.\_1 + total2.\_1, total1.\_2 + total2.\_2)

)

revenuePerDay.collect().foreach(println)

val avgRevenuePerDay = revenuePerDay.map(x => (x.\_1, x.\_2.\_1/x.\_2.\_2))

===========================================================

**CUSTOMER ID WITH MAX REVENUE**

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

val ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

val orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

val ordersParsedRDD = ordersRDD.map(rec => (rec.split(",")(0), rec))

val orderItemsParsedRDD = orderItemsRDD.map(rec => (rec.split(",")(1), rec))

val ordersJoinOrderItems = orderItemsParsedRDD.join(ordersParsedRDD)

val ordersPerDayPerCustomer = ordersJoinOrderItems.map(rec => ((rec.\_2.\_2.split(",")(1), rec.\_2.\_2.split(",")(2)), rec.\_2.\_1.split(",")(4).toFloat))

val revenuePerDayPerCustomer = ordersPerDayPerCustomer.reduceByKey((x, y) => x + y)

val revenuePerDayPerCustomerMap = revenuePerDayPerCustomer.map(rec => (rec.\_1.\_1, (rec.\_1.\_2, rec.\_2)))

val topCustomerPerDaybyRevenue = revenuePerDayPerCustomerMap.reduceByKey((x, y) => (if(x.\_2 >= y.\_2) x else y))

===========================================================

**USING REGULAR FUNCTION**

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

def findMax(x: (String, Float), y: (String, Float)): (String, Float) = {

if(x.\_2 >= y.\_2)

return x

else

return y

}

val topCustomerPerDaybyRevenue = revenuePerDayPerCustomerMap.reduceByKey((x, y) => findMax(x, y))

===========================================================

**FILTER DATA INTO SMALLER DATASET**

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

val ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

ordersRDD.filter(line => line.split(",")(3).equals("COMPLETE")).take(5).foreach(println)

ordersRDD.filter(line => line.split(",")(3).contains("PENDING")).take(5).foreach(println)

ordersRDD.filter(line => line.split(",")(0).toInt > 100).take(5).foreach(println)

ordersRDD.filter(line => line.split(",")(0).toInt > 100 || line.split(",")(3).contains("PENDING")).take(5).foreach(println)

ordersRDD.filter(line => line.split(",")(0).toInt > 1000 &&

(line.split(",")(3).contains("PENDING") || line.split(",")(3).equals("CANCELLED"))).

take(5).

foreach(println)

ordersRDD.filter(line => line.split(",")(0).toInt > 1000 &&

!line.split(",")(3).equals("COMPLETE")).

take(5).

foreach(println)

===========================================================

**#Check if there are any cancelled orders with amount greater than 1000$**

**#Get only cancelled orders**

**#Join orders and order items**

**#Generate sum(order\_item\_subtotal) per order**

**#Filter data which amount to greater than 1000$**

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

val ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

val orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

val ordersParsedRDD = ordersRDD.filter(rec => rec.split(",")(3).contains("CANCELED")).

map(rec => (rec.split(",")(0).toInt, rec))

val orderItemsParsedRDD = orderItemsRDD.

map(rec => (rec.split(",")(1).toInt, rec.split(",")(4).toFloat))

val orderItemsAgg = orderItemsParsedRDD.reduceByKey((acc, value) => (acc + value))

val ordersJoinOrderItems = orderItemsAgg.join(ordersParsedRDD)

ordersJoinOrderItems.filter(rec => rec.\_2.\_1 >= 1000).take(5).foreach(println)

==========================================================

**#Using SQL**

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

import org.apache.spark.sql.hive.HiveContext

val sqlContext = new HiveContext(sc)

sqlContext.sql("select \* from (select o.order\_id, sum(oi.order\_item\_subtotal) as order\_item\_revenue from orders o join order\_items oi on o.order\_id = oi.order\_item\_order\_id where o.order\_status = 'CANCELED' group by o.order\_id) q where order\_item\_revenue >= 1000").count()

**RANKING OR SORTING**

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

**Global sorting and ranking**

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

val orders = sc.textFile("/user/cloudera/sqoop\_import/orders")

orders.map(rec => (rec.split(",")(0).toInt, rec)).sortByKey().collect().foreach(println)

orders.map(rec => (rec.split(",")(0).toInt, rec)).sortByKey(false).take(5).foreach(println)

orders.map(rec => (rec.split(",")(0).toInt, rec)).top(5).foreach(println)

orders.map(rec => (rec.split(",")(0).toInt, rec)).

takeOrdered(5).

foreach(println)

orders.map(rec => (rec.split(",")(0).toInt, rec)).

takeOrdered(5)(Ordering[Int].reverse.on(x => x.\_1)).

foreach(println)

orders.takeOrdered(5)(Ordering[Int].on(x => x.split(",")(0).toInt)).foreach(println)

orders.takeOrdered(5)(Ordering[Int].reverse.on(x => x.split(",")(0).toInt)).foreach(println)

val products = sc.textFile("/user/cloudera/sqoop\_import/products")

val productsMap = products.map(rec => (rec.split(",")(1), rec))

val productsGroupBy = productsMap.groupByKey()

productsGroupBy.collect().foreach(println)

===========================================================

**DATA SORTED BY PRODUCT PRICE PER CATEGORY**

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

**#Get data sorted by product price per category**

**#You can use map or flatMap, if you want to see one record per line you need to use flatMap**

**#Map will return the list**

productsGroupBy.map(rec => (rec.\_2.toList.sortBy(k => k.split(",")(4).toFloat))).

take(100).

foreach(println)

productsGroupBy.map(rec => (rec.\_2.toList.sortBy(k => -k.split(",")(4).toFloat))).

take(100).

foreach(println)

productsGroupBy.flatMap(rec => (rec.\_2.toList.sortBy(k => -k.split(",")(4).toFloat))).

take(100).

foreach(println)

def getAll(rec: (String, Iterable[String])): Iterable[String] = {

return rec.\_2

}

productsGroupBy.flatMap(x => getAll(x)).collect().foreach(println)

===========================================================

**TOP N PRDUCTS BY PRICE IN EACH CATEGORY**

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

**#To get topN products by price in each category**

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

def getTopN(rec: (String, Iterable[String]), topN: Int): Iterable[String] = {

val x: List[String] = rec.\_2.toList.sortBy(k => -k.split(",")(4).toFloat).take(topN)

return x

}

val products = sc.textFile("/user/cloudera/sqoop\_import/products")

val productsMap = products.map(rec => (rec.split(",")(1), rec))

productsMap.groupByKey().flatMap(x => getTopN(x, 2)).collect().foreach(println)

===========================================================

**TOP N PRICED PRODUCTS BY CATEGORY**

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

**#To get topN priced products by category**

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

def getTopDenseN(rec: (String, Iterable[String]), topN: Int): Iterable[String] = {

var prodPrices: List[Float] = List()

var topNPrices: List[Float] = List()

var sortedRecs: List[String] = List()

for(i <- rec.\_2) {

prodPrices = prodPrices:+ i.split(",")(4).toFloat

}

topNPrices = prodPrices.distinct.sortBy(k => -k).take(topN)

sortedRecs = rec.\_2.toList.sortBy(k => -k.split(",")(4).toFloat)

var x: List[String] = List()

for(i <- sortedRecs) {

if(topNPrices.contains(i.split(",")(4).toFloat))

x = x:+ i

}

return x

}

productsMap.groupByKey().flatMap(x => getTopDenseN(x, 2)).collect().foreach(println)

==========================================================