package com.local.test  
import org.apache.spark.sql.SparkSession  
import org.apache.spark.sql.functions.\_  
object JsonReading {  
 def main(args: Array[String]): Unit = {  
 val spark=SparkSession.*builder*().appName("testqueryjson").master("local").getOrCreate()  
 import spark.implicits.\_  
 val df=spark.read.json("D:\\study\_material\\scalapractise\\csvfiles\\inputfile")  
 spark.sparkContext.setLogLevel("ERROR")  
 df.filter("name='harkirat'").show()

**filter:**

df.filter("name='harkirat'").show()

**GroupBy**

df.groupBy("city").count().show()

create temp table and temp table will be available as long as session is alive

df.createOrReplaceTempView("people")  
val sqldf=spark.sql("select \* from people")

Window Function ;

case class Salary(depName: String, empNo: Int, salary: Int)  
import *ss*.implicits.\_  
 val *empsalary* = *Seq*(  
 *Salary*("sales", 1, 5000),  
 *Salary*("personnel", 2, 3900),  
 *Salary*("sales", 3, 4800),  
 *Salary*("sales", 4, 4800),  
 *Salary*("personnel", 5, 3500),  
 *Salary*("develop", 7, 4200),  
 *Salary*("develop", 8, 6000),  
 *Salary*("develop", 9, 4500),  
 *Salary*("develop", 10, 5200),  
 *Salary*("develop", 11, 5200)).toDF.as[Salary]  
 *empsalary*.withColumn("avg",*row\_number*().over(Window.*partitionBy*("depName").  
 orderBy("depName"))).show()

Accumulator and Broadcast:

**Broadcast Variable :**

[**https://sparkbyexamples.com/spark/spark-broadcast-variables/**](https://sparkbyexamples.com/spark/spark-broadcast-variables/)

val *spark*=SparkSession.*builder*().appName("broadcast").master("local[\*]").getOrCreate()

  val *states*=*Map*(("NY","NewYork"),("CA","Delhi"),("FL","Florida"))

  val *statesbroadcast*=*spark*.sparkContext.broadcast(*states*)

  val *data* = *Seq*(("James","Smith","USA","CA"),

    ("Michael","Rose","USA","NE"),

    ("Robert","Williams","USA","CA"),

    ("Maria","Jones","USA","FL")

  )

  val *rdd*=*spark*.sparkContext.parallelize(*data*)

val *rdd2*= *rdd*.map{ x=>

    val state=x.\_4

    val statename=*statesbroadcast*.value.get(state).getOrElse("Not Available")

    (x.\_1,x.\_2,x.\_4,statename)

  }

*rdd2*.foreach(*println*)

}

**DataFrame:**

import *spark*.implicits.\_

val *rdd2*= *rdd*.map{ x=>

    val state=x.\_4

    val statename=*statesbroadcast*.value.get(state).getOrElse("Not Available")

    (x.\_1,x.\_2,x.\_4,statename)

  }.toDF()

*rdd2*.show()

}

**Accumulator:**

val *spark*=SparkSession.*builder*().appName("broadcast").master("local[\*]").getOrCreate()

val *order*=*spark*.sparkContext.textFile("/Users/mediakind/studymaterial/study\_material/CCA-175Spark/data-master/retail\_db/orders/part-00000")

val *ordercompleted*=*spark*.sparkContext.accumulator(0,"order completed")

val *orderfilterout*=*order*.filter( order => {

*ordercompleted* +=1

  order.split(",")(3)=="COMPLETE"

})

Unit Test Cases:

package ericsson.mediafirst.subscriberaccounts.reports  
  
import com.ericsson.mediafirst.data.categories.subscriber.spark.entities.{SkuReference, SubscriberAccount}  
import ericsson.mediafirst.subscriberaccounts.entity.{GroupAccountSchema, GroupSchema, RoutingGroup}  
import org.apache.spark.{SparkConf, SparkContext}  
import org.joda.time.{DateTime, DateTimeZone}  
import org.scalatest.{FunSuite, Ignore}  
  
@Ignore  
class SubscriberAccountTest extends FunSuite {  
  
 test("getReportOutput - 1") {  
  
 val tenantId = "default"  
  
 val conf = new SparkConf()  
 .setMaster("local")  
 .setAppName("Test2")  
 .set("spark.driver.allowMultipleContexts", "true")  
  
 val sc = new SparkContext(conf)  
 val (routingGroup1, routingGroup2, routingGroup3) = getRoutingGroups  
 val routingGroups = *List*(routingGroup2, routingGroup1, routingGroup3)  
 val routingGroupRdd = sc.parallelize(routingGroups)  
 .map(routingGroup => ((tenantId, routingGroup.accountId), routingGroup))  
  
 val (account1, account2, account3, account4) = getSubscriberAccounts  
 val accounts = *List*(account3, account1, account4, account2)  
 val accountRdd = sc.parallelize(accounts)  
  
 val skuReference = SkuReference(*Map*("mediafirstRequiredSKUDescriptions" -> "requiredDescription",  
 "mediafirstOptionalSKUDescriptions" -> "optionalDescription"))  
  
 val operator = "dev"  
  
 val fullEnvironment = "mr\_dev\_A"  
  
 // DateTime.now().withZone(DateTimeZone.UTC)  
 val loadDateTime = DateTime.*parse*("2018-07-06T12:26:58.000Z").withZone(DateTimeZone.*UTC*)  
 val indexTime = DateTime.*parse*("2018-07-06T00:00:00.000Z").withZone(DateTimeZone.*UTC*).toDate  
  
 val outputMaps = SubscriberAccountReport.*getReportOutput*(accountRdd, routingGroupRdd, skuReference, operator, fullEnvironment, loadDateTime)  
 .collect  
  
 val time = outputMaps.head.\_1.toString  
 val output1 = outputMaps(0).\_2 - "LogTime"  
 val output2 = outputMaps(1).\_2 - "LogTime"  
 val output3 = outputMaps(2).\_2 - "LogTime"  
 val output4 = outputMaps(3).\_2 - "LogTime"  
  
 assert(outputMaps.length == 4)  
 assert(time == "2018-07-06T00:00:00.000Z")  
  
 val expectedMap1 = *Map*("MediafirstRequiredSKUDescription" -> "requiredDescription", "groupType" -> "6", "Operator" -> "dev", "MediafirstRequiredSKU" -> "mediafirstRequiredSKUDescriptions", "routingGroup" -> "name1", "DvrNotificationsEnabled" -> true, "PbrOverride" -> true, "UserIds" -> *Set*("userId1", "userId2"), "Groups" -> *Set*("groupA", "groupB"), "IpAddress" -> "181.19.233.888", "groupId" -> "id1", "indexTime" -> indexTime, "AccountType" -> "type1", "AccountId" -> "accountId1", "uid" -> "FA7DD3241195B802F716A583342CB7F8", "Disabled" -> false, "fullEnvironment" -> "mr\_dev\_A", "release" -> "data1", "tenantId" -> "default", "MediafirstOptionalSKUDescription" -> *Set*(("mediafirstOptionalSKUDescriptions", "optionalDescription"), ("optionalSku2","")), "LegacyBranchId" -> "branchId1", "MediafirstOptionalSKUs" -> *Set*("mediafirstOptionalSKUDescriptions", "optionalSku2"), "DeviceIds" -> *Set*("deviceId1, deviceId2"), "DefaultUserId" -> "defaultuserid1", "LegacyAccountExternalId" -> "externalId1")  
 val expectedMap2 = *Map*("MediafirstRequiredSKUDescription" -> "requiredDescription", "groupType" -> "6", "Operator" -> "dev", "MediafirstRequiredSKU" -> "mediafirstRequiredSKUDescriptions", "routingGroup" -> "name2", "DvrNotificationsEnabled" -> true, "PbrOverride" -> true, "UserIds" -> *Set*("userId3", "userId5"), "Groups" -> *Set*("groupC", "groupD"), "IpAddress" -> "282.29.233.888", "groupId" -> "id2", "indexTime" -> indexTime, "AccountType" -> "type2", "AccountId" -> "accountId2", "uid" -> "68ACE9969C581F94EB256CDDB56FB147", "Disabled" -> false, "fullEnvironment" -> "mr\_dev\_A", "release" -> "data2", "tenantId" -> "default", "MediafirstOptionalSKUDescription" -> *Set*(("mediafirstOptionalSKUDescriptions", "optionalDescription"), ("optionalSku2","")), "LegacyBranchId" -> "branchId2", "MediafirstOptionalSKUs" -> *Set*("mediafirstOptionalSKUDescriptions", "optionalSku2"), "DeviceIds" -> *Set*("deviceId3, deviceId5"), "DefaultUserId" -> "defaultuserid2", "LegacyAccountExternalId" -> "externalId2")  
 val expectedMap3 = *Map*("MediafirstRequiredSKUDescription" -> "requiredDescription", "groupType" -> "", "Operator" -> "dev", "MediafirstRequiredSKU" -> "mediafirstRequiredSKUDescriptions", "routingGroup" -> "", "DvrNotificationsEnabled" -> true, "PbrOverride" -> true, "UserIds" -> *Set*("userId8", "userId9"), "Groups" -> *Set*("groupG", "groupH"), "IpAddress" -> "185.19.155.188", "groupId" -> "", "indexTime" -> indexTime, "AccountType" -> "type5", "AccountId" -> "accountId5", "uid" -> "21C9A6F2D027A1243E60DA1689A87207", "Disabled" -> false, "fullEnvironment" -> "mr\_dev\_A", "release" -> "", "tenantId" -> "default", "MediafirstOptionalSKUDescription" -> *Set*(("mediafirstOptionalSKUDescriptions", "optionalDescription"), ("optionalSku5","")), "LegacyBranchId" -> "branchId5", "MediafirstOptionalSKUs" -> *Set*("mediafirstOptionalSKUDescriptions", "optionalSku5"), "DeviceIds" -> *Set*("deviceId8, deviceId9"), "DefaultUserId" -> "defaultuserid5", "LegacyAccountExternalId" -> "externalId5")  
 val expectedMap4 = *Map*("MediafirstRequiredSKUDescription" -> "requiredDescription", "groupType" -> "6", "Operator" -> "dev", "MediafirstRequiredSKU" -> "mediafirstRequiredSKUDescriptions", "routingGroup" -> "name3", "DvrNotificationsEnabled" -> true, "PbrOverride" -> true, "UserIds" -> *Set*("userId6", "userId7"), "Groups" -> *Set*("groupE", "groupF"), "IpAddress" -> "183.19.133.188", "groupId" -> "id3", "indexTime" -> indexTime, "AccountType" -> "type3", "AccountId" -> "accountId3", "uid" -> "717559B7C913A947946EF75C8B2E35CF", "Disabled" -> false, "fullEnvironment" -> "mr\_dev\_A", "release" -> "data3", "tenantId" -> "default", "MediafirstOptionalSKUDescription" -> *Set*(("mediafirstOptionalSKUDescriptions", "optionalDescription"), ("optionalSku3","")), "LegacyBranchId" -> "branchId3", "MediafirstOptionalSKUs" -> *Set*("mediafirstOptionalSKUDescriptions", "optionalSku3"), "DeviceIds" -> *Set*("deviceId6, deviceId7"), "DefaultUserId" -> "defaultuserid3", "LegacyAccountExternalId" -> "externalId3")  
  
 assert(output1 == expectedMap1)  
 assert(output2 == expectedMap2)  
 assert(output3 == expectedMap3)  
 assert(output4 == expectedMap4)  
 sc.stop  
 }  
  
 test("getAccountGroupData - 1") {  
  
 val conf = new SparkConf()  
 .setMaster("local")  
 .setAppName("Test2")  
 .set("spark.driver.allowMultipleContexts", "true")  
  
 val sc = new SparkContext(conf)  
  
 val tenantId = "default"  
  
 val (groupSchema1, groupSchema2, groupSchema3, groupSchema4, groupSchema5, groupSchema6, groupSchema7) = getGroups  
 val groups = *List*(groupSchema3, groupSchema7, groupSchema4, groupSchema2, groupSchema6, groupSchema5, groupSchema1)  
 val groupRdd = sc.parallelize(groups)  
 .map(group => (tenantId, group))  
  
 val (groupAccount1, groupAccount2, groupAccount3, groupAccount4) = getGroupAccounts  
 val groupAccounts = *List*(groupAccount4, groupAccount2, groupAccount1, groupAccount3)  
 val groupAccountRdd = sc.parallelize(groupAccounts)  
 .map(groupAccount => (tenantId, groupAccount))  
  
 val routingGroups = SubscriberAccountReport.*getAccountGroupData*(groupRdd, groupAccountRdd)  
 .collect  
  
 assert(routingGroups.length == 3)  
 val routingGroup1 = routingGroups(0).\_2  
 val routingGroup2 = routingGroups(1).\_2  
 val routingGroup3 = routingGroups(2).\_2  
  
 val (expectedRoutingGroup1, expectedRoutingGroup2, expectedRoutingGroup3) = getRoutingGroups  
 assert(routingGroup1 == expectedRoutingGroup3)  
 assert(routingGroup2 == expectedRoutingGroup2)  
 assert(routingGroup3 == expectedRoutingGroup1)  
  
 sc.stop  
 }  
  
 test("joinGroupAccountGroup - 1") {  
  
 val conf = new SparkConf()  
 .setMaster("local")  
 .setAppName("Test2")  
 .set("spark.driver.allowMultipleContexts", "true")  
  
 val sc = new SparkContext(conf)  
  
 val tenantId = "default"  
  
 val (groupSchema1, groupSchema2, groupSchema3, groupSchema4, groupSchema5, groupSchema6, groupSchema7) = getGroups  
 val groups = *List*(groupSchema3, groupSchema7, groupSchema4, groupSchema2, groupSchema6, groupSchema5, groupSchema1)  
 val groupRdd = sc.parallelize(groups)  
 .map(group => (tenantId, group))  
  
 val (groupAccount1, groupAccount2, groupAccount3, groupAccount4) = getGroupAccounts  
 val groupAccounts = *List*(groupAccount2, groupAccount3, groupAccount4, groupAccount1)  
 val groupAccountRdd = sc.parallelize(groupAccounts)  
 .map(groupAccount => (tenantId, groupAccount))  
  
 val routingGroups = SubscriberAccountReport.*joinGroupAccountGroup*(groupRdd, groupAccountRdd)  
 .collect  
  
 assert(routingGroups.length == 3)  
 val routingGroup1 = routingGroups(0).\_2  
 val routingGroup2 = routingGroups(1).\_2  
 val routingGroup3 = routingGroups(2).\_2  
  
 val (expectedRoutingGroup1, expectedRoutingGroup2, expectedRoutingGroup3) = getRoutingGroups  
 assert(routingGroup1 == expectedRoutingGroup3)  
 assert(routingGroup2 == expectedRoutingGroup2)  
 assert(routingGroup3 == expectedRoutingGroup1)  
  
 sc.stop  
 }  
  
 def getGroups: (GroupSchema, GroupSchema, GroupSchema, GroupSchema, GroupSchema, GroupSchema, GroupSchema) = {  
  
 val groupSchema1 = *GroupSchema*(grouptype = 6, querytype = 0, queryvalue = "", id = "id1", data = *Option*("data1"),  
 name = *Option*("name1"), subgroupid = *Option*("subGroup1"))  
  
 val groupSchema2 = *GroupSchema*(grouptype = 6, querytype = 0, queryvalue = "", id = "id2", data = *Option*("data2"),  
 name = *Option*("name2"), subgroupid = *Option*("subGroup2"))  
  
 val groupSchema3 = *GroupSchema*(grouptype = 6, querytype = 0, queryvalue = "", id = "id3", data = *Option*("data3"),  
 name = *Option*("name3"), subgroupid = *Option*("subGroup3"))  
  
 val groupSchema4 = *GroupSchema*(grouptype = 6, querytype = 0, queryvalue = "", id = "id4", data = *Option*("data4"),  
 name = *Option*("name4"), subgroupid = *Option*("subGroup4"))  
  
 val groupSchema5 = *GroupSchema*(grouptype = 6, querytype = 8, queryvalue = "", id = "id5", data = *Option*("data5"),  
 name = *Option*("name5"), subgroupid = *Option*("subGroup5"))  
  
 val groupSchema6 = *GroupSchema*(grouptype = 6, querytype = 0, queryvalue = "queryvalue", id = "id6", data = *Option*("data6"),  
 name = *Option*("name6"), subgroupid = *Option*("subGroup6"))  
  
 val groupSchema7 = *GroupSchema*(grouptype = 3, querytype = 0, queryvalue = "", id = "id7", data = *Option*("data7"),  
 name = *Option*("name7"), subgroupid = *Option*("subGroup7"))  
  
 (groupSchema1, groupSchema2, groupSchema3, groupSchema4, groupSchema5, groupSchema6, groupSchema7)  
 }  
  
 def getGroupAccounts: (GroupAccountSchema, GroupAccountSchema, GroupAccountSchema, GroupAccountSchema) = {  
 val groupAccount1 = *GroupAccountSchema*(grouptype = 6, groupid = "id1", accountid = "accountId1")  
 val groupAccount2 = *GroupAccountSchema*(grouptype = 6, groupid = "id2", accountid = "accountId2")  
 val groupAccount3 = *GroupAccountSchema*(grouptype = 6, groupid = "id3", accountid = "accountId3")  
 val groupAccount5 = *GroupAccountSchema*(grouptype = 8, groupid = "id5", accountid = "accountId5")  
  
 (groupAccount1, groupAccount2, groupAccount3, groupAccount5)  
 }  
  
 def getRoutingGroups: (RoutingGroup, RoutingGroup, RoutingGroup) = {  
 val routingGroup3 = *RoutingGroup*(accountId = "accountId3", groupId = "id3", groupType = 6, release = "data3", routingGroup = "name3", tenantId = "default")  
 val routingGroup2 = *RoutingGroup*(accountId = "accountId2", groupId = "id2", groupType = 6, release = "data2", routingGroup = "name2", tenantId = "default")  
 val routingGroup1 = *RoutingGroup*(accountId = "accountId1", groupId = "id1", groupType = 6, release = "data1", routingGroup = "name1", tenantId = "default")  
 (routingGroup1, routingGroup2, routingGroup3)  
 }  
  
 def getSubscriberAccounts: (SubscriberAccount, SubscriberAccount, SubscriberAccount, SubscriberAccount) = {  
  
 val account1 = SubscriberAccount(accountId = "accountId1", defaultuserid = "defaultuserid1", deviceids = *Set*("deviceId1, deviceId2"),  
 disabled = false, dvrnotificationsenabled = true, groups = *Set*("groupA", "groupB"), ipaddress = "181.19.233.888",  
 legacyaccountexternalid = "externalId1", legacybranchid = "branchId1", time = "", pbroverride = true,  
 userids = *Set*("userId1", "userId2"), profileids = *Set*("profileId1, profileId2"), accounttype = "type1",  
 mediafirstRequiredSKU = "mediafirstRequiredSKUDescriptions",  
 mediafirstOptionalSKUs = *Set*("mediafirstOptionalSKUDescriptions", "optionalSku2"), *Map*("Created" -> DateTime.*parse*("2018-08-17T15:22:20.871Z")),tenantId = "default")  
  
 val account2 = SubscriberAccount(accountId = "accountId2", defaultuserid = "defaultuserid2", deviceids = *Set*("deviceId3, deviceId5"),  
 disabled = false, dvrnotificationsenabled = true, groups = *Set*("groupC", "groupD"), ipaddress = "282.29.233.888",  
 legacyaccountexternalid = "externalId2", legacybranchid = "branchId2", time = "", pbroverride = true,  
 userids = *Set*("userId3", "userId5"), profileids = *Set*("profileId3, profileId5"), accounttype = "type2",  
 mediafirstRequiredSKU = "mediafirstRequiredSKUDescriptions",  
 mediafirstOptionalSKUs = *Set*("mediafirstOptionalSKUDescriptions", "optionalSku2"), *Map*("Created" -> DateTime.*parse*("2018-08-17T15:22:20.871Z")),tenantId = "default")  
  
 val account3 = SubscriberAccount(accountId = "accountId3", defaultuserid = "defaultuserid3", deviceids = *Set*("deviceId6, deviceId7"),  
 disabled = false, dvrnotificationsenabled = true, groups = *Set*("groupE", "groupF"), ipaddress = "183.19.133.188",  
 legacyaccountexternalid = "externalId3", legacybranchid = "branchId3", time = "", pbroverride = true,  
 userids = *Set*("userId6", "userId7"), profileids = *Set*("profileId6, profileId7"), accounttype = "type3",  
 mediafirstRequiredSKU = "mediafirstRequiredSKUDescriptions",  
 mediafirstOptionalSKUs = *Set*("mediafirstOptionalSKUDescriptions", "optionalSku3"), *Map*("Created" -> DateTime.*parse*("2018-08-17T15:22:20.871Z")),tenantId = "default")  
  
 val account5 = SubscriberAccount(accountId = "accountId5", defaultuserid = "defaultuserid5", deviceids = *Set*("deviceId8, deviceId9"),  
 disabled = false, dvrnotificationsenabled = true, groups = *Set*("groupG", "groupH"), ipaddress = "185.19.155.188",  
 legacyaccountexternalid = "externalId5", legacybranchid = "branchId5", time = "", pbroverride = true,  
 userids = *Set*("userId8", "userId9"), profileids = *Set*("profileId8, profileId9"), accounttype = "type5",  
 mediafirstRequiredSKU = "mediafirstRequiredSKUDescriptions",  
 mediafirstOptionalSKUs = *Set*("mediafirstOptionalSKUDescriptions", "optionalSku5"), *Map*("Created" -> DateTime.*parse*("2018-08-17T15:22:20.871Z")),tenantId = "default")  
(account1, account2, account3, account5)  
 }  
}