# CCA175

## Cheat sheet

Use Zooming (Ctrl + ) and Copy-Paste ( ctrl+shift+c and ctrl+shift+v)

spark-shell --master yarn --packages com.databricks:spark-avro\_2.10:2.0.1

spark2-shell --master yarn --packages com.databricks:spark-avro\_2.11:4.0.0

--conf spark.ui.port=15674

--num-executors 10

--executor-cores 2

--executor-memory 3G

### Reading files

# To read any text file (compressed / uncompressed into an RDD)

rdd = sc.textFile("…path")

#To read any sequence file (compressed / uncompressed into an RDD)

rdd = sc.sequenceFile("…path")

#To read a parquet file (compressed/uncompressed):

df = sqlContext.read.parquet("…path")

#To read a orc file (compressed/uncompressed):

df = sqlContext.read.orc("…path")

#To read a json file (compressed/uncompressed):

df = sqlContext.read.json("…path")

#To read a text file (compressed/uncompressed) to a data frame:

df = sqlContext.read.text("…path")

#To read an avro file (compressed/uncompressed):

df = sqlContext.load("…path","com.databricks.spark.avro")

or

import com.databricks.spark.avro.\_;

df = sqlContext.read.avro("…path");

### Writing files

#To write an RDD into text file to HDFS. By default compressionClassCodec=None. This means no compression.

Available codecs: BZip2Codec or GzipCodec or SnappyCodec.

rdd.saveAsTextFile("…path", classOf[org.apache.hadoop.io.compress.SnappyCodec])

#To write an RDD into a sequence file:

rdd.saveAsSequenceFile("…path", Some(classOf[org.apache.hadoop.io.compress.GzipCodec]))

#save DF with options:

df.write.format("csv").mode("overwrite").option("sep", "#")

.partitionBy("column\_name").save("/tmp/output/csv")

#To write data frame in parquet, avro, json, orc with/without compression (default: snappy):

#1. Set the parquet compression. In place of "gzip", use "snappy", "lzo", "uncompressed" or "gzip"

sqlContext.setConf("spark.sql.parquet.compression.codec","gzip")

#2. Write the file now in parquet format. mode can be "overwrite" or "append" or "error"

df.write.parquet("…path", mode="overwrite"))

#1. Set avro compression # Can be "snappy", "deflate" or "uncompressed"

sqlContext.setConf("spark.sql.avro.compression.codec","snappy")

#2. Write in avro format now #

df.format('com.databricks.spark.avro').save("…path")

or

import com.databricks.spark.avro.\_;   
df.write.avro("…path");

#I think by default the data is compressed by snappy format for ORC

df.write.orc("…path")

#Write the JSON file without compression

df.toJSON.saveAsTextFile("…path")

# Write the JSON file with compression

df.toJSON.saveAsTextFile("…path", classOf[org.apache.hadoop.io.compress.GzipCodec])

df.write.option("compression", "gzip").json("…path") // for Spark 2

get # of partitions:

df.rdd.getNumPartitions

#Write RDD and DF with desired number of partitions ##

# - For RDD:

rdd.coalesce(2).saveAsTextFile("…path")

# - For DF:

df.rdd.coalesce(2).toDF()

or

val singlePartitionDF = df.coalesce(1)

### Transformation

# RDD to DF

var orders\_rdd=sc.textFile("/user/training/data/retail\_db/orders")

var orders\_df=orders\_rdd.map(x =>

{ var d = x.split(",");

(d(0).toInt,d(1),d(2).toInt,d(3))

}).toDF("order\_id","order\_date","order\_customer\_id","order\_status")

# Join DF, optional left join

var order\_joined = orders\_df.join(order\_items\_df,orders\_df("order\_id") === order\_items\_df("order\_item\_order\_id"), "left")

### Dealing with Hive

Create Hive context (in Spark 2.x it's not needed as it is merged into spark.sqlContext)

var hc = new org.apache.spark.sql.hive.HiveContext(sc)

# view registered SQL views:

spark.catalog.listTables.show

# **multi**-line SQL using triple quotes:

spark.sql("""select \* from m""")

# save DF as hive table in Spark (deprecated)

someDF.registerTempTable("temptablename")

sqlContext.sql("select \* from temptablename").show

val someDF=sqlContext.sql("select \* from temptablename")

val options = Map("path" -> hiveTablePath) //optional hive path

someDF.write.mode("overwrite").options(options).saveAsTable("dbname.hiveTable")

# Connect to hive? (just use "hive" command)

beeline -u jdbc:hive2://localhost:10000/default -n root -p xxxxxx

# Run query from command line

hive -e "USE retail\_db; SHOW TABLES; SELECT \* FROM orders;"

# Create a hive table referencing the avro file

use retail\_hive\_db;

create table orders\_a(order\_id int, order\_time timestamp, cust\_id int, order\_status string) stored as AVRO;

// in Spark:

orders\_df.registerTempTable("orders\_df")

sqlContext.sql("""INSERT into retail\_hive\_db.orders\_a select \* from orders\_df""")

SELECT \* from retail\_hive\_db.orders\_a

# Create Hive database

create database my\_retail\_db;

use my\_retail\_db;

# Create table

DROP TABLE IF EXISTS orders;

CREATE EXTERNAL TABLE orders (

order\_id INT,

order\_date TIMESTAMP,

order\_customer\_id INT,

order\_status STRING

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

LINES TERMINATED BY '\n'

stored as AVRO (stored as textfile, stored as PARQUET)

LOCATION '/user/hive/warehouse/my\_retail\_db/orders';

# Load data from DB

sqoop import --connect … --table orders

--target-dir /user/hive/warehouse/my\_retail\_db/orders -m 1 --as-avrodatafile

# Load data from local file

load data local inpath "/home/training/data/retail\_db/orders" into table orders;

# Describe table

describe formatted orders

# Show the content of HDFS directory:

hdfs dfs -ls /<path>

or recursively hadoop fs -ls -R /<path>

# view compressed HDFS file

hdfs dfs -text /user/training/text\_gzip/part-m-00000.gz

# Download a file to Local from HDFS:

hdfs dfs -get /user/hive/warehouse/my\_retail\_db/orders/part-m-00000.avro

# Check avro schema from local or directly from HDFS file

avro-tools getschema part-m-00000.avro > orders.avsc

*or from* hdfs://localhost/user/hive/warehouse/my\_retail\_db/orders/part-m-00000.avro

# Upload a file to HDFS:

hdfs dfs -copyFromLocal orders.avsc /user/hive/warehouse/my\_retail\_db/orders\_schema/

# Create managed table based on a new avro schema

create table orders\_avro

stored as avro

tblproperties('avro.schema.url'='/user/hive/warehouse/my\_retail\_db/orders\_schema/orders.avsc');

#Load into managed table from HDFS path

load data inpath "/user/hive/warehouse/my\_retail\_db/orders" into table orders\_avro;

#Partitioning

Create table … partitioned by (order\_status string)

#Bucketing

Create table … clustered by (order\_date) sorted by (order\_date ) into 4 buckets

### Sqoop

sqoop import --help

# sqoop execute query

sqoop eval \

--connect jdbc:mysql://localhost/db:3306 --username root --password <password> \

--query "SELECT \* FROM employee LIMIT 3"

# sqoop list tables

sqoop list-tables \

--connect jdbc:mysql://localhost/db:3306 --username root --password <password>

# sqoop import all tables with compression

sqoop import-all-tables

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username root --password cloudera

--warehouse-dir /user/hive/warehouse/retail\_stage.db

--compress

--compression-codec snappy (or org.apache.hadoop.io.compress.SnappyCodec)

--as-avrodatafile (--as-textfile, --as-parquetfile, --as-sequencefile)

--delete-target-dir

-m 1;

#sqoop import columns from one table, with WHERE clause

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username root --password cloudera

--table departments

--columns "department\_id,department\_name"

--target-dir /user/cloudera/departments

--where "product\_id > 10000"

# sqoop import into hive using --query (query required $CONDITIONS and --split-by) . Notice **single** quotes.

sqoop import

--connect jdbc:mysql://localhost/retail\_db

--username root --password cloudera

--as-parquetfile

--hive-import

--hive-database "test"

--hive-table "queryParquet"

--hive-overwrite

--create-hive-table

--target-dir /user/cloudera/name

--query 'SELECT \* FROM orders WHERE order\_id < 10 AND $CONDITIONS'

--split-by 'order\_id'

# Access MySQL, if needed, to create table

mysql -h localhost -u root -p

create table order\_item\_dtl

(order\_id Int,

order\_date date,

order\_status varchar(45),

order\_item\_subtotal Float

);

# Export

sqoop export

--connect "jdbc:mysql://quickstart:3306/retail\_db"

--username retail\_dba

--password cloudera

--table products\_export

--export-dir "/user/hive/warehouse/cca175/products\_export"

# export - only update target, no insert

--update-mode updateonly

--update-key department\_id

# export - insert and update

--update-mode allowinsert

--update-key department\_id

# other params (\001 - default hive delimiter - can also be entered as Ctrl+V+A in command line)

--input-fields-terminated-by "\001"

--input-lines-terminated-by "\n"

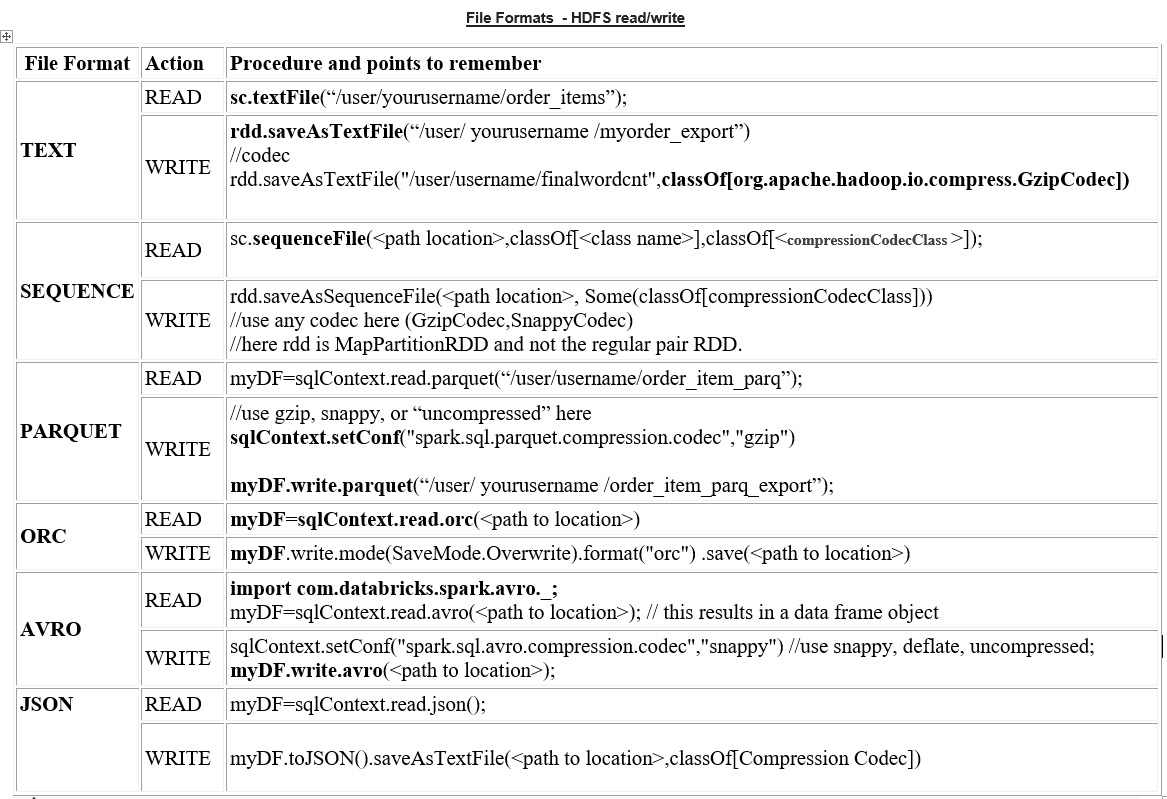
--input-null-string ""

--input-null-non-string -999

Uses MySQL's default delimiter set: fields: , lines: \n escaped-by: \ optionally-enclosed-by: '

--mysql-delimiters

## File Formats



## Exercises

### Problem 1

Import orders table from mysql (db: retail\_db , user : retail\_user , password : xxxx)

Import only records that are in "COMPLETE" status

Import all columns other than customer id

Save the imported data as text and tab delimitted in this hdfs location /user/cloudera/problem1/

sqoop import   
--connect jdbc:mysql://localhost:3306/retail\_db   
--username retail\_dba    
--password cloudera   
--table orders   
--columns order\_id,order\_date,order\_status   
--where "order\_status like 'COMPLETE'"   
--target-dir /user/cloudera/problem1   
--as-textfile   
--fields-terminated-by '\t'

### Problem 2

Import orders table from mysql (db: retail\_db , user : retail\_user , password : xxxx)

Import all records and columns from Orders table

Save the imported data as text and tab delimitted in this hdfs location /user/cloudera/problem2/

sqoop import   
--connect jdbc:mysql://localhost:3306/retail\_db   
--username retail\_dba    
--password cloudera   
--table orders   
--target-dir /user/cloudera/problem2   
--fields-terminated-by '\t'   
--as-textfile

### Problem 3

Export orders data into mysql

Input Source : /user/cloudera/problem2/

Target Table : Mysql . DB = retail\_export . Table Name : mock\_orders

> mysql -h localhost:3306 -u retail\_user -p  
mysql> create database retail\_export; use retail\_export;  
mysql> create table mock\_orders as select \* from retail\_db.orders limit 1;  
mysql> truncate mock\_orders;

sqoop export   
--connect jdbc:mysql://localhost:3306/retail\_export   
--username retail\_dba    
--password cloudera   
--table mock\_orders   
--export-dir /user/cloudera/problem2   
--input-fields-terminated-by '\t'

### Problem 4

Read table populated from Problem 3 (mock\_orders )

Produce output in this format (2 fields) , sort by order count in descending and save it as avro with snappy compression in hdfs location /user/cloudera/problem4/avro-snappy

ORDER\_STATUS : ORDER\_COUNT

COMPLETE 54

CANCELLED 89

INPROGRESS 23

sqoop import   
--connect jdbc:mysql://localhost:3306/retail\_export   
--username retail\_dba    
--password cloudera   
--query "select order\_status,count(1) order\_count from mock\_orders where \$CONDITIONS group by order\_status order by order\_count desc "   
--as-avrodatafile   
--compress   
--compression-codec snappy   
--target-dir /user/cloudera/problem4/avro-snappy   
-m 1

### Problem 5

Import orders table from mysql (db: retail\_db , user : retail\_user , password : xxxx)

Import all records and columns from Orders table

Save the imported data as avro and snappy compression in hdfs location /user/cloudera/problem5-avro-snappy/

Read above hdfs data

Consider orders only in "COMPLETE" status and order id between 1000 and 50000 (1001 to 49999)

Save the output (only 2 columns orderid and orderstatus) in parquet format with gzip compression in location /user/cloudera/problem5-parquet-gzip/

Advance : Try if you can save output only in 2 files (Tip : use coalesce(2))

sqoop import   
--connect jdbc:mysql://localhost:3306/retail\_db   
--username retail\_dba    
--password cloudera   
--table orders   
--as-avrodatafile   
--compress   
--compression-codec snappy   
--target-dir /user/cloudera/problem5-avro-snappy/

val ordersDF= sqlContext.read.format("com.databricks.spark.avro").load("/user/cloudera/problem5-avro-snappy")  
ordersDF.registerTempTable("orders")  
val result = sqlContext.sql("select order\_id,order\_status from orders where order\_status like 'COMPLETE' and order\_id > 1000 and order\_id < 50000")  
// alternative:  
val result = ordersDF.filter("order\_id >1000 and order\_id<50000 and order\_status like 'COMPLETE'").select("order\_id","order\_status")  
  
sqlContext.setConf("spark.sql.parquet.compression.codec","gzip")  
result.coalesce(2).write.parquet("/user/cloudera/problem5-parquet-gzip")

Version 2: compression with deflate level 5

sqlContext.setConf("spark.sql.avro.comression.codec", "deflate")

sqlContext.setConf("spark.sql.avro.deflate.level", "5")

### Problem 6

Import orders table from mysql (db: retail\_db , user : retail\_user , password : xxxx)

Import all records and columns from Orders table

Save the imported data as text and tab delimitted in this hdfs location /user/cloudera/problem6/orders/

Import order\_items table from mysql (db: retail\_db , user : retail\_user , password : xxxx)

Import all records and columns from Order\_items table

Save the imported data as text and tab delimitted in this hdfs location /user/cloudera/problem6/order-items/

Read orders data from above HDFS location

Read order items data form above HDFS location

Produce output in this format (price and total should be treated as decimals)

Consider only CLOSED & COMPLETE orders

ORDER\_ID ORDER\_ITEM\_ID PRODUCT\_PRICE ORDER\_SUBTOTAL ORDER\_TOTAL

( ORDER\_TOTAL = combined total price for this order )

Save above output as ORC in hive table "mock\_orderdetails"

(Tip : Try saving into hive table from DF directly without explicit table creation manually)

sqoop import   
--connect jdbc:mysql://localhost:3306/retail\_db   
--username retail\_dba    
--password cloudera   
--table orders   
--as-textfile   
--fields-terminated-by '\t'   
--target-dir /user/cloudera/problem6/orders/

sqoop import   
--connect jdbc:mysql://localhost:3306/retail\_db   
--username retail\_dba    
--password cloudera   
--table order\_items   
--as-textfile   
--fields-terminated-by '\t'   
--target-dir /user/cloudera/problem6/order\_items/

*// Load files using RDD*val orders = sc.textFile("/user/cloudera/problem6/orders/")  
val order\_items = sc.textFile("/user/cloudera/problem6/order\_items")  
var ordersMap = ordersData.map(\_.split("\t")).map(o=>(o(0),o(1),o(2),o(3))).

toDF("order\_id","order\_date","order\_customer\_id","order\_status")

var orderitemsMap = ordersitemsData.map(\_.split("\t")).map(o=>(o(0),o(1),o(2),o(3),o(4),o(5))).

toDF("order\_item\_id","order\_item\_order\_id","order\_item\_product\_id","order\_item\_quantity",

"order\_item\_subtotal","order\_item\_product\_price")

*// Alternative dataframe load using "csv":*val ordersDF = spark.read.option("sep", "\t").csv("/user/cloudera/problem6/orders")  
.toDF("order\_id", "order\_date","order\_customer\_id","order\_status")  
val order\_itemsDF = spark.read.option("sep", "\t").csv("/user/cloudera/problem6/order\_items")  
.toDF("order\_item\_id", "order\_item\_order\_id", "order\_item\_product\_id", "order\_item\_quantity", "order\_item\_subtotal","order\_item\_product\_price")  
  
ordersDF.registerTempTable("orders")  
order\_itemsDF.registerTempTable("order\_items")

*// Alternative join without sql via join() function:*

val dfJoined = ordersDF.join(orderItemsDF, orderItemsDF("order\_item\_order\_id") === ("ordersDF.order\_id"))

dfJoined.registerTempTable("orderRevenue")

val result = sqlContext.sql("select order\_id, order\_item\_id,cast (order\_item\_product\_price as decimal (10,2)),cast(order\_item\_subtotal as decimal(10,2)), sum(cast(order\_item\_subtotal as decimal(10,2))) over (partition by order\_id) ORDER\_TOTAL from order\_items oi join orders o on oi.order\_item\_order\_id = o.order\_id where order\_status in ('CLOSED','COMPLETE') ")  
  
*// With option "path" it creates external Hive table. Otherwise - managed table in default location*result.write.format("orc").option("path", "/user/cloudera/problem6/hiveorders").mode("overwrite")  
.saveAsTable("mock\_orderdetails")

### Problem 7

Import order\_items table from mysql (db: retail\_db , user : retail\_user , password : xxxx)

Import all records and columns from Order\_items table

Save the imported data as parquet in this hdfs location /user/cloudera/problem7/order-items/

Import products table from mysql (db: retail\_db , user : retail\_user , password : xxxx)

Import all records and columns from products table

Save the imported data as avro in this hdfs location /user/cloudera/problem7/products/

Read above orderitems and products from HDFS location

Produce this output :

ORDER\_ITEM\_ORDER\_ID PRODUCT\_ID PRODUCT\_NAME PRODUCT\_PRICE ORDER\_SUBTOTAL

Save above output as avro snappy in hdfs location /user/cloudera/problem7/output-avro-snappy/

sqoop import   
--connect jdbc:mysql://localhost:3306/retail\_db   
--username retail\_dba    
--password cloudera   
--table order\_items   
--as-parquetfile   
--target-dir /user/cloudera/problem7/order\_items/

sqoop import   
--connect jdbc:mysql://localhost:3306/retail\_db   
--username retail\_dba    
--password cloudera   
--table products   
--as-avrodatafile   
--target-dir /user/cloudera/problem7/products/

order\_items = sqlContext.read.parquet("/user/cloudera/problem7/order\_items")  
order\_items.registerTempTable("order\_items")

val products = sqlContext.read.format("com.databricks.spark.avro").load("/user/cloudera/problem7/products/")  
products.registerTempTable("products")

val result = sqlContext.sql("select order\_item\_id,product\_id,cast(product\_price as decimal), cast(order\_item\_subtotal as decimal(10,2)) from products p join order\_items oi on p.product\_id = oi.order\_item\_product\_id ")  
sqlContext.setConf("spark.sql.avro.compression.codec","snappy")  
result.write.format("com.databricks.spark.avro").save("/user/cloudera/problem7/output-avro-snappy/")

### Problem 8

Read order item from /user/cloudera/problem7/order-items/

Read products from /user/cloudera/problem7/products/

Produce output that shows product id and total no. of orders for each product id.

Output should be in this format… sorted by order count descending

If any product id has no order then order count for that product id should be "0"

PRODUCT\_ID PRODUCT\_PRICE ORDER\_COUNT

Output should be saved as sequence file with Key=ProductID , Value = PRODUCT\_ID|PRODUCT\_PRICE|ORDER\_COUNT (pipe separated)

val order\_items = sqlContext.read.parquet("/user/cloudera/problem7/order\_items")  
order\_items.registerTempTable("order\_items")

import com.databricks.spark.avro.\_; *// if import namespace then can load using .avro*

val products = sqlContext.read.avro("/user/cloudera/problem7/products")

products.registerTempTable("products")

val result = sqlContext.sql("select product\_id,product\_price,count(order\_item\_id) order\_count from products p left outer join order\_items oi on p.product\_id = oi.order\_item\_product\_id group by product\_id,product\_price ")  
val resultMap = result.rdd.map(x=>(x(0),x(0).toString +"|"+x(1)+"|"+x(2)))

resultMap.saveAsSequenceFile("/user/cloudera/problem8/orders\_sequence")

### Problem 9

Import orders table from mysql (db: retail\_db , user : retail\_user , password : xxxx)

Import all records and columns from Orders table

Save the imported data as avro in this hdfs location /user/cloudera/problem9/orders-avro/

Read above Avro orders data

Convert to JSON

Save JSON text file in hdfs location /user/cloudera/problem9/orders-json/

Read json data from /user/cloudera/problem9/orders-json/

Consider only "COMPLETE" orders.

Save orderid and order status (just 2 columns) as JSON text file in location /user/cloudera/problem9/orders-mini-json/

sqoop import   
--connect jdbc:mysql://localhost:3306/retail\_db   
--username retail\_dba    
--password cloudera   
--table orders   
--as-avrodatafile   
--target-dir /user/cloudera/problem9/orders-avro/

val orders = sqlContext.read.format("com.databricks.spark.avro").load("/user/cloudera/problem9/orders-avro/")  
orders.write.json("/user/cloudera/problem9/orders-json/")  
val ojson=sqlContext.read.json("/user/cloudera/problem9/orders-json/")  
ojson.registerTempTable("ojson")  
val reslt = sqlContext.sql("select order\_id, order\_status from ojson where order\_status like 'COMPLETE' ")  
reslt.write.json("/user/cloudera/problem9/orders-mini-json/")

## arun-teaches-u-tech.blogspot.com

### Scenario 1

sqoop import \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username retail\_dba \

--password cloudera \

--table orders \

--compress \

--compression-codec org.apache.hadoop.io.compress.SnappyCodec \

--target-dir /user/cloudera/problem1/orders \

--as-avrodatafile;

Spark:

import com.databricks.spark.avro.\_;

var ordersDF = sqlContext.read.avro("/user/cloudera/problem1/orders");

var joinedOrderDataDF = ordersDF

.join(orderItemDF,ordersDF("order\_id")===orderItemDF("order\_item\_order\_id"))

* with dataframes:

import org.apache.spark.sql.functions.\_;

var dataFrameResult = joinedOrderDataDF.

groupBy(to\_date(from\_unixtime(col("order\_date")/1000)).alias("order\_formatted\_date"),col("order\_status")).

agg(round(sum("order\_item\_subtotal"),2).alias("total\_amount"),countDistinct("order\_id").alias("total\_orders")).

orderBy(col("order\_formatted\_date").desc,col("order\_status"),col("total\_amount").desc,col("total\_orders"));

* or with sql:

joinedOrderDataDF.registerTempTable("order\_joined");

var sqlResult = sqlContext.sql("select to\_date(from\_unixtime(cast(order\_date/1000 as bigint))) as order\_formatted\_date, order\_status, cast(sum(order\_item\_subtotal) as DECIMAL (10,2)) as total\_amount, count(distinct(order\_id)) as total\_orders from order\_joined group by order\_formatted\_date, order\_status order by order\_formatted\_date desc,order\_status,total\_amount desc, total\_orders");

…continue

sqlContext.setConf("spark.sql.parquet.compression.codec","gzip");

dataFrameResult.write.parquet("/user/cloudera/problem1/result4a-gzip");

sqlContext.setConf("spark.sql.parquet.compression.codec","snappy");

dataFrameResult.write.parquet("/user/cloudera/problem1/result4a-snappy");

dataFrameResult.write.format("csv").mode("overwrite").option("sep", ",").save("/user/cloudera/problem1/ result4a-csv")

sqoop export --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username retail\_dba --password cloudera \

--table result \

--export-dir "/user/cloudera/problem1/result4a-csv" \

--columns "order\_date,order\_status,total\_amount,total\_orders"

### Scenario 2

sqoop import \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username retail\_dba \

--password cloudera \

--table products \

--as-textfile \

--target-dir /user/cloudera/products \

--fields-terminated-by '|';

hadoop fs -chmod 765 /user/cloudera/problem2/products/\* (permissions: owner/group/other rwe/rw/re)

> var products = sc.textFile("/user/cloudera/products").map(x=> {var d = x.split('|'); (d(0).toInt,d(1).toInt,d(2).toString,d(3).toString,d(4).toFloat,d(5).toString)});

> case class Product(productID:Integer, productCategory: Integer, productName: String, productDesc:String, productPrice:Float, productImage:String);

> var productsDF=products.map(x=> Product(x.\_1,x.\_2,x.\_3,x.\_4,x.\_5,x.\_6)).toDF();

Data Frame Api:

> import org.apache.spark.sql.functions.\_

> var dataFrameResult = productsDF.filter("productPrice < 100")

.groupBy(col("productCategory")).agg(max(col("productPrice")).alias("max\_price"),countDistinct(col("productID")).alias("tot\_products"),

round(avg(col("productPrice")),2).alias("avg\_price"),

min(col("productPrice")).alias("min\_price")).orderBy(col("productCategory"));

Spark SQL:

> productsDF.registerTempTable("products");

> var sqlResult = sqlContext.sql("select productCategory, max(productPrice) as maximum\_price, count(distinct(productID)) as total\_products, cast(avg(productPrice) as decimal(10,2)) as average\_price, min(productPrice) as minimum\_price from products where productPrice < 100 group by productCategory order by productCategory desc");

> import com.databricks.spark.avro.\_;

> sqlContext.setConf("spark.sql.avro.compression.codec","snappy")

> dataFrameResult.write.avro("/user/cloudera/problem2/products/result-df");

### Scenario 3

sqoop import-all-tables \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username retail\_dba --password cloudera \

--warehouse-dir /user/hive/warehouse/retail\_stage.db \

--compress \

--compression-codec snappy \

--as-avrodatafile

-m 1;

hadoop fs -get /user/hive/warehouse/retail\_stage.db/orders/part-m-00000.avro

avro-tools getschema part-m-00000.avro > orders.avsc

hadoop fs -mkdir /user/hive/schemas

hadoop fs - mkdir /user/hive/schemas/order

hadoop fs -copyFromLocal orders.avsc /user/hive/schemas/order

hive

create external table orders\_sqoop

STORED AS AVRO

LOCATION '/user/hive/warehouse/retail\_stage.db/orders'

TBLPROPERTIES ('avro.schema.url'='/user/hive/schemas/order/orders.avsc');

create table orders\_avro

> (order\_id int,

> order\_date date,

> order\_customer\_id int,

> order\_status string)

> partitioned by (order\_month string)

> STORED AS AVRO;

insert overwrite table orders\_avro partition (order\_month)

select order\_id, to\_date(from\_unixtime(cast(order\_date/1000 as int))), order\_customer\_id, order\_status, substr(from\_unixtime(cast(order\_date/1000 as int)),1,7) as order\_month from default.orders\_sqoop;

# Edit Avro Schema (add 2 fields):

gedit orders.avsc

,{

"name" : "order\_style",

"type" : [ "null", "string" ],

"default" : null,

"columnName" : "order\_style",

"sqlType" : "12"

}, {

"name" : "order\_zone",

"type" : [ "null", "int" ],

"default" : null,

"columnName" : "order\_zone",

"sqlType" : "4"

}

hadoop fs -copyFromLocal **-f** orders.avsc /user/hive/schemas/order/orders.avsc

### Scenario 4

sqoop import --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" --password cloudera --username retail\_dba --table orders --as-avrodatafile--target-dir /user/cloudera/problem5/avro -m 1

var dataFile = sqlContext.read.avro("/user/cloudera/problem5/avro");

# save using snappy compression as parquet file

sqlContext.setConf("spark.sql.parquet.compression.codec","snappy");

dataFile.repartition(1).write.parquet("/user/cloudera/problem5/parquet-snappy-compress");

# save using gzip compression as text file

dataFile.map(x=>x(0)+"\t"+x(1)+"\t"+x(2)+"\t"+x(3)).saveAsTextFile("/user/cloudera/problem5/text-gzip-compress",classOf[org.apache.hadoop.io.compress.GzipCodec]);

# save using snappy compression as text file. Below may fail in some cloudera VMS.

# If the spark command fails use the sqoop command to accomplish the problem.

dataFile.map(x=> x(0)+"\t"+x(1)+"\t"+x(2)+"\t"+x(3)).saveAsTextFile("/user/cloudera/problem5/text-snappy-compress",classOf[org.apache.hadoop.io.compress.SnappyCodec]);

sqoop import --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username retail\_dba --password cloudera --table orders --as-textfile -m1

--target-dir user/cloudera/problem5/text-snappy-compress

--compress --compression-codec org.apache.hadoop.io.compress.SnappyCodec

# read previous and save the data to hdfs using no compression as parquet file

var parquetDataFile = sqlContext.read.parquet("/user/cloudera/problem5/parquet-snappy-compress")

sqlContext.setConf("spark.sql.parquet.compression.codec","uncompressed");

parquetDataFile.write.parquet("/user/cloudera/problem5/parquet-no-compress");

# save the data to hdfs using snappy compression as avro file

parquetDataFile.write.avro("/user/cloudera/problem5/avro-snappy");

# read previous and save the data to hdfs using gzip compression as json file

var avroData = sqlContext.read.avro("/user/cloudera/problem5/avro-snappy");

avroData.toJSON.saveAsTextFile("/user/cloudera/problem5/json-gzip",classOf[org.apache.hadoop.io.compress.GzipCodec]);

# read previous and save the data to as comma separated text using gzip compression

var jsonData = sqlContext.read.json("/user/cloudera/problem5/json-gzip");

jsonData.map(x=>x(0)+","+x(1)+","+x(2)+","+x(3)).saveAsTextFile("/user/cloudera/problem5/csv-gzip",classOf[org.apache.hadoop.io.compress.GzipCodec])

#To read the sequence file you need to understand the sequence getter for the key and value class to be used

//In a new terminal Get the Sequence file to local file system

hadoop fs -get /user/cloudera/problem5/sequence/part-00000

//read the first 300 characters to understand the two classes to be used.

cut -c-300 part-00000

// read file in Spark

var seqData = sc.sequenceFile("/user/cloudera/problem5/sequence/", classOf[org.apache.hadoop.io.Text],classOf[org.apache.hadoop.io.Text]);

// write using no compression as ORC file

seqData.map(x=>{var d = x.\_2.toString.split("\t"); (d(0),d(1),d(2),d(3))}).toDF().write.orc("/user/cloudera/problem5/orc");

### Scenario 6

sqoop import-all-tables --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username retail\_dba --password cloudera

--warehouse-dir /user/hive/warehouse/problem6.db

--hive-import --hive-database problem6 --create-hive-table --as-textfile;

var hc = new org.apache.spark.sql.hive.HiveContext(sc);

// 3. Rank products within department by price and order by department asc and rank desc

var hiveResult = hc.sql(

"select d.department\_id, p.product\_id, p.product\_name, p.product\_price,

rank() over (partition by d.department\_id order by p.product\_price) as product\_price\_rank,

dense\_rank() over (partition by d.department\_id order by p.product\_price) as product\_dense\_price\_rank

from products p inner join categories c on c.category\_id = p.product\_category\_id inner join departments d on c.category\_department\_id = d.department\_id order by d.department\_id, product\_price\_rank desc, product\_dense\_price\_rank ");

// 4. Find top 10 customers with most unique product purchases. If more than one customer has the same number of product purchases, then the customer with the lowest customer\_id will take precedence

var hiveResult2 = hc.sql("select c.customer\_id, c.customer\_fname, count(distinct(oi.order\_item\_product\_id)) unique\_products

from customers c inner join orders o on o.order\_customer\_id = c.customer\_id inner join order\_items oi on o.order\_id = oi.order\_item\_order\_id

group by c.customer\_id, c.customer\_fname

order by unique\_products desc, c.customer\_id limit 10")

// 5. On dataset from step 3, apply filter such that only products less than 100 are extracted

hiveResult.registerTempTable("product\_rank\_result\_temp");

hc.sql("select \* from product\_rank\_result\_temp where product\_price < 100").show();

// 6. On dataset from step 4, extract details of products purchased by top 10 customers which are priced at less than 100 USD per unit

var topCustomers = hc.sql("select c.customer\_id, c.customer\_fname, count(distinct(oi.order\_item\_product\_id)) unique\_products

from customers c inner join orders o on o.order\_customer\_id = c.customer\_id inner join order\_items oi on o.order\_id = oi.order\_item\_order\_id

group by c.customer\_id, c.customer\_fname

order by unique\_products desc, c.customer\_id limit 10");

topCustomers.registerTempTable("top\_cust");

var topProducts = hc.sql("select distinct p.\* from products p inner join order\_items oi on oi.order\_item\_product\_id = p.product\_id inner join orders o on o.order\_id = oi.order\_item\_order\_id inner join top\_cust tc on o.order\_customer\_id = tc.customer\_id where p.product\_price < 100");

// Store the result of 5 and 6 in new meta store tables within hive

hc.sql("create table problem6.product\_rank\_result as select \* from product\_rank\_result\_temp where product\_price < 100");

hc.sql("create table problem6.top\_products as select distinct p.\* from products p inner join order\_items oi on oi.order\_item\_product\_id = p.product\_id inner join orders o on o.order\_id = oi.order\_item\_order\_id inner join top\_cust tc on o.order\_customer\_id = tc.customer\_id where p.product\_price < 100");

## itversity

http://discuss.itversity.com/c/certifications/spark-exercises

### Exercise 02

Get the customers who have not placed any orders, sorted by customer\_lname and then customer\_fname

Target Columns: customer\_lname, customer\_fname

Number of files - 1

Target Directory: /user/<YOUR\_USER\_ID>/solutions/solutions02/inactive\_customers

Target File Format: TEXT

Target Delimiter: comma (", ")

import scala.io.Source

val ordersRaw = Source.fromFile("/data/retail\_db/orders/part-00000").getLines.toList

val orders = sc.parallelize(ordersRaw)

val customersRaw = Source.fromFile("/data/retail\_db/customers/part-00000").getLines.toList

val customers = sc.parallelize(customersRaw)

Core API:

val ordersMap = orders.

map(order => (order.split(",")(2).toInt, 1))

val customersMap = customers.

map(c => (c.split(",")(0).toInt, (c.split(",")(2), c.split(",")(1))))

val customersLeftOuterJoinOrders = customersMap.leftOuterJoin(ordersMap)

val inactiveCustomersSorted = customersLeftOuterJoinOrders.

filter(t => t.\_2.\_2 == None).

map(rec => rec.\_2).

sortByKey()

inactiveCustomersSorted.

map(rec => rec.\_1.\_1 + ", " + rec.\_1.\_2).

saveAsTextFile("/user/dgadiraju/solutions/solutions02/inactive\_customers")

DataFrames:

val ordersDF = orders.

map(o => o.split(",")(2).toInt).

toDF("order\_customer\_id")

val customersDF = customers.

map(c => (c.split(",")(0).toInt, c.split(",")(1), c.split(",")(2))).

toDF("customer\_id", "customer\_fname", "customer\_lname")

ordersDF.registerTempTable("orders\_dg")

customersDF.registerTempTable("customers\_dg")

sqlContext.setConf("spark.sql.shuffle.partitions", "1")

sqlContext.

sql("select customer\_lname, customer\_fname " +

"from customers\_dg left outer join orders\_dg " +

"on customer\_id = order\_customer\_id " +

"where order\_customer\_id is null " +

"order by customer\_lname, customer\_fname").

rdd.

map(rec => rec.mkString(", ")).

saveAsTextFile("/user/dgadiraju/solutions/solutions02/inactive\_customers")

### Exercise 15

Get rank of each category by revenue with in each department generated from all the transactions

Display the results by deparment\_name and rank in ascending order

hiveContext.sql("select d\_name, c\_name, revenue, rank()

over (partition by d\_name order by revenue desc) as rank\_ from

( select d.department\_name as d\_name, c.category\_name as c\_name, sum(oi.order\_item\_subtotal) as revenue from departments d

join categories c on ( c.category\_department\_id = d.department\_id)

join products p on ( p.product\_category\_id = c.category\_id)

join order\_items oi on ( oi.order\_item\_product\_id = p.product\_id)

group by d.department\_name, c.category\_name order by d.department\_name ) q1")

### Using spark-submit

# totalrevenuedaily.sh:

spark-submit --class retail.dataframes.TotalRevenueDaily \

--master yarn \

--conf spark.ui.port=54123 \

sparkdemo\_2.10-1.0.jar \

/public/retail\_db /user/dgadiraju/totalrevenuedaily prod

#totalrevenueperday.scala:

|  |
| --- |
| package retail.dataframes  import com.typesafe.config.ConfigFactory |
| import org.apache.spark.SparkConf |
| import org.apache.spark.SparkContext |
| import org.apache.hadoop.fs.\_ |
| import org.apache.spark.sql.\_ |
| import org.apache.spark.sql.functions.\_ |
| object TotalRevenueDaily { |
| def main(args: Array[String]) { |
| val appConf = ConfigFactory.load() |
| val conf = new SparkConf(). |
| setAppName("Total Revenue - Daily - Data Frames"). |
| setMaster(appConf.getConfig(args(2)).getString("executionMode")) |
| val sc = new SparkContext(conf) |
| val sqlContext = new SQLContext(sc) |
| sqlContext.setConf("spark.sql.shuffle.partitions", "2") |
|  |
| import sqlContext.implicits.\_ |
| val inputPath = args(0) |
| val outputPath = args(1) |
| val fs = FileSystem.get(sc.hadoopConfiguration) |
| val inputPathExists = fs.exists(new Path(inputPath)) |
| val outputPathExists = fs.exists(new Path(outputPath)) |
| if (!inputPathExists) { |
| println("Input Path does not exists") |
| return |
| } |
| if (outputPathExists) { |
| fs.delete(new Path(outputPath), true) |
| } |
| val ordersDF = sc.textFile(inputPath + "/orders"). |
| map(rec => { |
| val a = rec.split(",") |
| Orders(a(0).toInt, a(1).toString(), a(2).toInt, a(3).toString()) |
| }).toDF() |
| val orderItemsDF = sc.textFile(inputPath + "/order\_items"). |
| map(rec => { |
| val a = rec.split(",") |
| OrderItems( |
| a(0).toInt, |
| a(1).toInt, |
| a(2).toInt, |
| a(3).toInt, |
| a(4).toFloat, |
| a(5).toFloat) |
| }).toDF() |
| val ordersFiltered = ordersDF. |
| filter(ordersDF("order\_status") === "COMPLETE") |
| val ordersJoin = ordersFiltered.join(orderItemsDF, |
| ordersFiltered("order\_id") === orderItemsDF("order\_item\_order\_id")) |
|  |
| ordersJoin. |
| groupBy("order\_date"). |
| agg(sum("order\_item\_subtotal")). |
| sort("order\_date"). |
| rdd. |
| saveAsTextFile(outputPath) |
| } |
| } |