**SCD Type 1 & 2 implementation with spark and scala**

Contents

[Using Timestamp Datatype: 2](#_Toc352366)

[Implementing SCD2 4](#_Toc352367)

[Pivoting 12](#_Toc352368)

[Retrieving top 2 values from each field 13](#_Toc352369)

[Window Function 14](#_Toc352370)

[Explode function to split a String Column 16](#_Toc352371)

# Using Timestamp Datatype:

scala> :paste

// Entering paste mode (ctrl-D to finish)

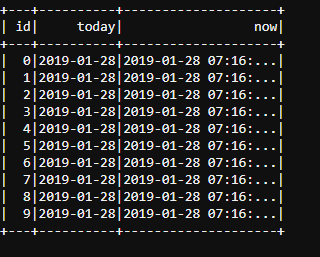
import org.apache.spark.sql.functions.{current\_date, current\_timestamp}

valdateDF = sqlContext.range(10)

.withColumn("today", current\_date())

.withColumn("now", current\_timestamp())

dateDF.show



dateDF.withColumn("Year",year(dateDF("today"))).show

dateDF.withColumn("Week",weekofyear(dateDF("today"))).show

dateDF.withColumn("Month",month(dateDF("today"))).show

dateDF.withColumn("Month",dayofmonth(dateDF("today"))).show

dateDF.withColumn("cur\_minute",minute(col("now"))).show

dateDF.withColumn("cur\_second",second(col("now"))).show

dateDF.withColumn("new\_added\_months\_date", expr("add\_months(today, 7)")).show

Useful Date functions with Dataframes:

|  |  |
| --- | --- |
| **Function** | **Description** |
| date\_sub | To subtract specific number of days from a give date |
| date\_add | To add specific number of days from a give date |
| datediff | To subtract two dates |
| months\_between | To find the number of months between two dates |
| to\_date | To conver a string literal to a date |
| year | To extract year from date |
| weekofyear | To extract week of year |
| month | To extract month from date |
| dayofmonth | To extract day of month |
| minute | To extract minute from timestamp |
| second | To extract second from timestamp |
| add\_months | To add months to date |

To change the date format:

valdateFormat = "yyyy-dd-MM"

sqlContext.range(1).select(

to\_date(from\_unixtime(unix\_timestamp(lit("20171211"), "yyyyMMdd"), dateFormat)).alias("date")).show

# Implementing SCD2

Initial load:

// brief description about steps followed to build SCD2.

// 1. yelp\_data\_actHist = Active records from History file which are not found in current file.

// 2. yelp\_data\_intersect = Active records from History file for which there are no updates in current file (same record is available).

// 3. yelp\_data\_expired = Inactive records from History file.

// 4. yelp\_data\_upnew\_date = Set of Updated and new records from current file.

// 5. yelp\_data\_upexp\_date = End date updated records from history as the updates found for them in current file.

InputFile:

hadoop fs -cat yelp\_user\_20190129.csv

user\_id,name,yelping\_since,useful,elite,average\_stars

zzNKg1,Jenna,2011-12-20,0,None,5.0

zzNKg2,Jenna,2011-12-20,2500,2015 2016 2017,4.5

zzNKg3,Jenna,2011-12-20,3000,2015 2016 2018,3.5

zzNKg4,Jenna,2011-12-20,3400,2015 2017 2018,4.0

import org.apache.hadoop.fs.{FileSystem, Path}

import org.apache.hadoop.fs.\_

import org.apache.spark.sql.\_

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

import scala.io.Source

import java.text.SimpleDateFormat

valproc\_dt = "20190129"

valinputFormat = new SimpleDateFormat("yyyyMMdd")

valreqFormat = new SimpleDateFormat("yyyy-MM-dd")

// Convert the effective and expiry date in yyyy-MM-dd format

valeff\_dt = reqFormat.format(inputFormat.parse(proc\_dt))

valexp\_dt = reqFormat.format(reqFormat.parse("9999-12-31"))

valhistPath = new Path("/user/respond474/yelp\_hist/")

valhistTabPath= "/user/respond474/yelp\_hist/"

valinput\_file = s"/user/respond474/yelp\_user\_20190129.csv"

val fs = FileSystem.get(sc.hadoopConfiguration)

sqlContext.setConf("spark.sql.shuffle.partitions", "10")

import sqlContext.implicits.\_

// Check if the HIST table directory present for first run, else create directory

if (!fs.exists(histPath)) {

println("History table directory does not exist, creating directory")

fs.mkdirs(histPath)

} else {

println("History table directory exist")

}

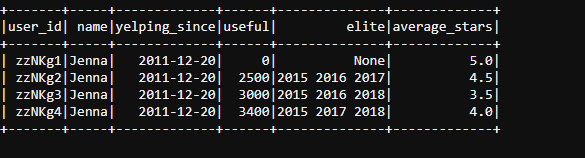
valyelp\_data = sqlContext.read.format("com.databricks.spark.csv").

option("header", "true").

option("inferSchema", "true").

option("quote", "\"").

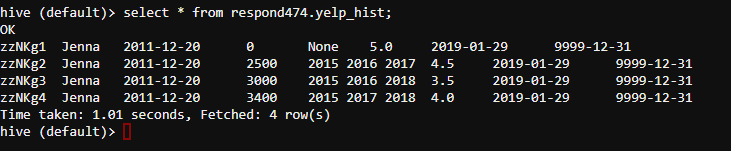
load(input\_file)



valyelp\_data\_hist = yelp\_data.withColumn("eff\_dt",to\_date(from\_unixtime(unix\_timestamp(lit(eff\_dt),"yyyy-MM-dd"), "yyyy-MM-dd"))).withColumn("exp\_dt",to\_date(from\_unixtime(unix\_timestamp(lit(exp\_dt),"yyyy-MM-dd"), "yyyy-MM-dd")))

import org.apache.spark.sql.SaveMode

yelp\_data\_hist.coalesce(2).write.mode(SaveMode.Overwrite).format("orc").saveAsTable("respond474.yelp\_hist")



Incremental load:

hadoop fs -cat yelp\_user\_20190130.csv

user\_id,name,yelping\_since,useful,elite,average\_stars

zzNKg1,Jenna,2011-12-20,3400,2015 2018 2018,8.0

BBBBB1,Jenna,2011-12-20,3400,2015 2018 2018,12.0

valstgPath = new Path("/user/respond474/yelp\_hist\_stg/")

valhistPath = new Path("/apps/hive/warehouse/respond474.db/yelp\_hist")

valhistTabPath = "/apps/hive/warehouse/respond474.db/yelp\_hist"

valstgTabPath = "/user/respond474/yelp\_hist\_stg/"

valinput\_file = s"/user/respond474/yelp\_user\_20190130.csv"

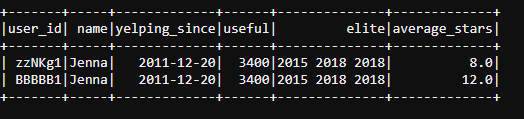
valyelp\_data\_stg = sqlContext.read.format("com.databricks.spark.csv").

option("header", "true").

option("inferSchema", "true").

option("quote", "\"").

load(input\_file)



valyelp\_data\_hist = sqlContext.read.orc(histTabPath)

yelp\_data\_hist.cache

valyelp\_data\_expired = yelp\_data\_hist.as("hist").filter($"exp\_dt" !== "9999-12-31").select($"hist.\*")

valyelp\_data\_intersect = yelp\_data\_hist.as("hist").filter($"exp\_dt"==="9999-12-31").

join(yelp\_data\_stg.as("stg"), Seq("user\_id"), "inner").

where($"hist.exp\_dt" === "9999-12-31"

&& $"hist.useful" === $"stg.useful"

&& $"hist.average\_stars" === $"stg.average\_stars"

&& $"hist.yelping\_since" === $"stg.yelping\_since"

&& $"hist.name" === $"stg.name"

&& $"hist.elite" === $"stg.elite"). select($"hist.\*")

//New or updated records

valyelp\_data\_upnew = yelp\_data\_stg.as("stg").

join(yelp\_data\_hist.as("hist").filter($"exp\_dt"==="9999-12-31"), $"hist.user\_id" === $"stg.user\_id", "left").

where(

($"hist.user\_id".isNull) ||

($"hist.user\_id" === $"stg.user\_id" &&

(($"hist.useful" !== $"stg.useful" ) ||

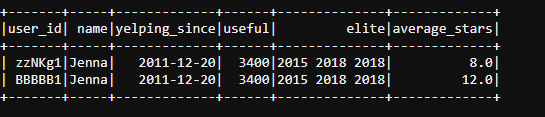
($"hist.yelping\_since" !== $"stg.yelping\_since" ) ||

($"hist.name" !== $"stg.name" ) ||

($"hist.average\_stars" !== $"stg.average\_stars")

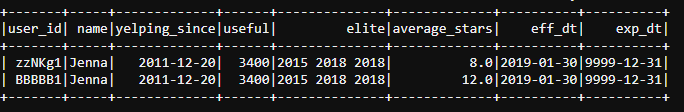
))).select($"stg.\*")

valyelp\_data\_upnew.show()



valyelp\_data\_upnew\_date = yelp\_data\_upnew.withColumn("eff\_dt",to\_date(from\_unixtime(unix\_timestamp(lit(eff\_dt), "yyyy-MM-dd"), "yyyy-MM-dd"))).

withColumn("exp\_dt",to\_date(from\_unixtime(unix\_timestamp(lit("9999-12-31"),"yyyy-MM-dd"), "yyyy-MM-dd")))



// Get old updated user\_id from the HIST table and set expiry date i.e. (2099-12-31)

valyelp\_data\_upexp = yelp\_data\_hist.as("hist").filter($"exp\_dt"==="9999-12-31").

join(yelp\_data\_stg.as("stg"), Seq("user\_id"), "inner").

where(

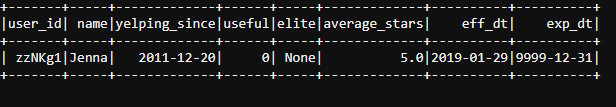
($"hist.useful" !== $"stg.useful" ) ||

($"hist.yelping\_since" !== $"stg.yelping\_since" ) ||

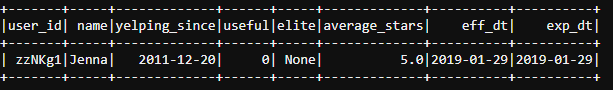
($"hist.name" !== $"stg.name" ) ||

($"hist.average\_stars" !== $"stg.average\_stars")

).select($"hist.\*")



valyelp\_data\_upexp\_date = yelp\_data\_upexp.withColumn("exp\_dt",to\_date(from\_unixtime(unix\_timestamp(lit(exp\_dt),"yyyy-MM-dd"), "yyyy-MM-dd")))



// Active user\_id from the HIST table that are not in STG

valyelp\_data\_actHist = yelp\_data\_hist.as("hist").filter($"exp\_dt"==="9999-12-31").

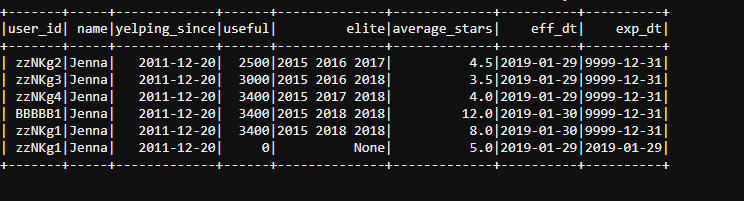
join(yelp\_data\_stg.as("stg"), $"hist.user\_id" === $"stg.user\_id", "left").

where( $"stg.user\_id".isNull).select($"hist.\*")

// Merge data from all the above dataframes to create historical table data

valdfUnion = Seq(yelp\_data\_actHist,yelp\_data\_intersect,yelp\_data\_expired,yelp\_data\_upnew\_date,yelp\_data\_upexp\_date)

valyelp\_data\_all = dfUnion.reduce(\_.unionAll(\_))



yelp\_data\_all.write.mode(SaveMode.Overwrite).format("orc").save(stgTabPath)

yelp\_data\_all.cache

if (fs.exists(histPath)) {

println("History table directory exist, deleting the old table data")

fs.delete(histPath)

} else {

println("History table directory does not exist")

}

if (fs.exists(stgPath)) {

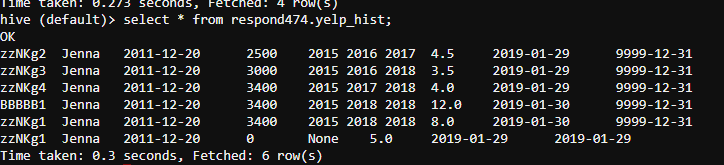
println("Staging directory is loaded with updated data, ")

fs.rename(stgPath,histPath)

} else {

println("Staging directorydoes not exist")

}



# Pivoting

Input file [pivot\_input.csv]

id,tag,value

1,US,50

1,UK,100

1,Can,125

2,US,75

2,UK,150

2,Can,175

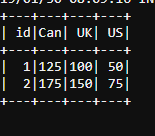
valinputDF = sqlContext.read.format("com.databricks.spark.csv").

option("header", "true").

option("inferSchema", "true").

load("pivot\_input.csv")

inputDF.groupBy("id").pivot("tag").agg(sum("value")).show



# Retrieving top 2 values from each field

input\_file(mixName.txt):

mixname,soda,formula1,formula2,formula3

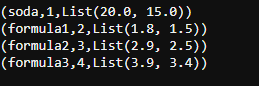
m1,10,1.5,2.5,3.4

m2,20,1.3,2.3,3.1

m3,15,1.8,2.9,3.0

m4,12,1.1,2.1,3.9

Steps to transform input as below:



valdf = sc.textFile("/user/respond474/mixName.txt").filter(x=> (x.split(",")(1) != "soda")).map(x => {val v = x.split(","); (v(0), v(1).toDouble, v(2).toDouble, v(3).toDouble, v(4).toDouble)}).toDF("mixname","soda","formula1","formula2","formula3")

valrowLength = df.schema.length

valrankRDD = df.flatMap( row => Range(1, rowLength).map(i => (i, row.getDouble(i))))

valrankSorted = rankRDD.groupByKey.map(x => (x.\_1, x.\_2.toArray.sortWith(\_>\_))).sortByKey(true)

valfinal\_res = rankSorted.map(x => (x.\_1, x.\_2.zipWithIndex)).map(row => (row.\_1, Range(0, 2).map(i => row.\_2(i)).map(x=> x.\_1).toList))

val header = sc.textFile("/user/respond474/mixName.txt").first.split(",")

val final\_res\_1 = final\_res.map(row => (header(row.\_1), row.\_1, row.\_2))

# Window Function

case class Salary(depName: String, empNo: Long, salary: Long)

valempsalary = Seq(

Salary("sales", 1, 5000),

Salary("personnel", 2, 3900),

Salary("sales", 3, 4800),

Salary("sales", 4, 4800),

Salary("personnel", 5, 3500),

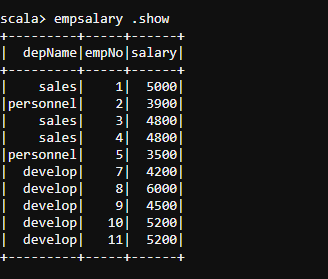
Salary("develop", 7, 4200),

Salary("develop", 8, 6000),

Salary("develop", 9, 4500),

Salary("develop", 10, 5200),

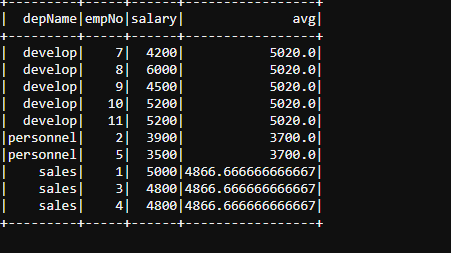
Salary("develop", 11, 5200)).toDF



import org.apache.spark.sql.expressions.Window

valbyDepName = Window.partitionBy('depName)

empsalary.withColumn("avg", avg('salary) over byDepName).show

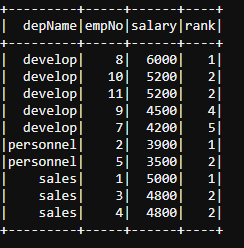


/////////////////////////////////////////////////////////////////////

valbyDepnameSalaryDesc = Window.partitionBy('depname).orderBy('salary desc)

valrankByDepname = rank().over(byDepnameSalaryDesc)

empsalary.select('\*, rankByDepname as 'rank).show



//////////////////////////////////////////////////////////////////////

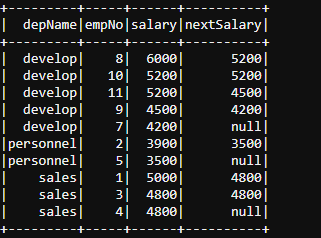
valpartitionWindow = Window.partitionBy($"depname").orderBy($"salary".desc).rowsBetween(1,1)

The window frame is defined as starting from 1 (one row after the current row) and ending at 1 (one row after the current row)

Preceding row can be indicated as -1 (one row before the current row)

valnext\_val = max($"salary").over(partitionWindow)

empsalary.select($"\*", next\_val as 'nextSalary).show



# Explode function to split a String Column

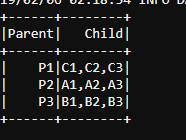
Input file(child\_parent.txt):

P1|C1,C2,C3

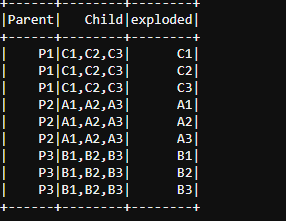
P2|A1,A2,A3

P3|B1,B2,B3

valdf = sc.textFile("/user/respond474/child\_parent.txt").map(x => (x.split('|')(0),x.split('|')(1))).toDF("Parent", "Child")



df.withColumn("splitted", split(col("Child"), ",")).withColumn("exploded", explode(col("splitted"))).drop(col("splitted")).show



It is also possible to add depth using spark-sql.

val df1 = df.withColumn("splitted", split(col("Child"), ","))

df1.registerTempTable("df\_tab")

sqlContext.sql("""

select

Parent, Child, exploded.dIndex + 1 as depth, exploded.splitted

from

df\_tab

lateral view posexplode(df\_tab.splitted) exploded as dIndex, splitted

""").show

