1. **Design**

To solve the Music Data Analysis Requirement the following modules are used

1. **LoadHBaseTables –** This is used for loading the lookup data from csv files to the relevant tables in Hbase. Hbase tables used are as below:

* **StationIdGeoCd –**  This HBase table is used for storing mapping between stationId and GeoCd
* **SongArtist -** This HBase table is used for storing mapping beween Song and Artist
* **UserArtist –** This HBase table is used for storing mapping beween userId and List of artists he follows
* **UserSubscritpion-** This HBase table is used for storing mapping beween userId and subscription start timestamp and end timestamp

1. **MusicDataProcessorApp -** This is the main application for processing music data. It in turn calls multiple submodules as below
2. **WebMusicDataProcessor –** This is the class for processing music data stored in /data/web/file-1.xml folder and use processData method to return dataframes
3. **MobileMusicDataProcessor -** This is the class for processing music data stored in /data/mob/file.txt folder and use processData method to return dataframes
4. **MusicDataEnricher -** This is the used for enriching and validating the datasets. New columns are added to dataframe for the processed data
5. **MusicDataPopulateMapFromLookupTables -** This is the used for populating maps from HBase tables used for lookup
6. **MusicDataAnalyzer - -** This is used for analyzing the data in dataframes and store the results in HDFS tables
7. **Implementation**

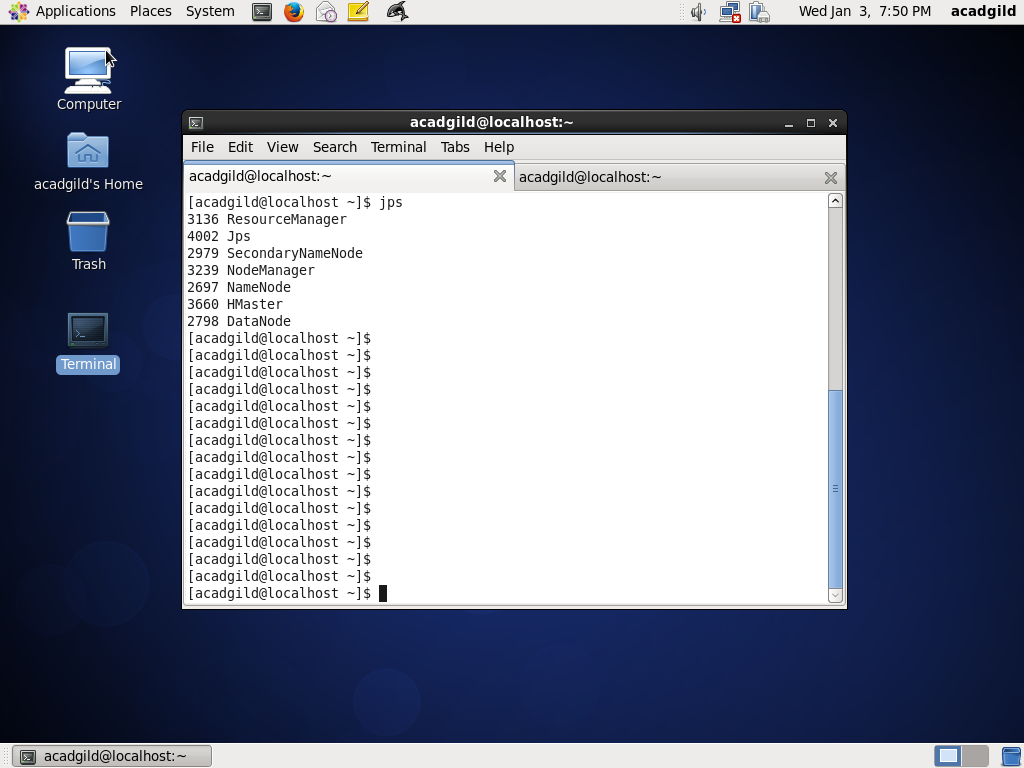
**MODULE1: LoadHBaseTables: Loading HBase Tables with Lookup Data**

Step1: Start all the services

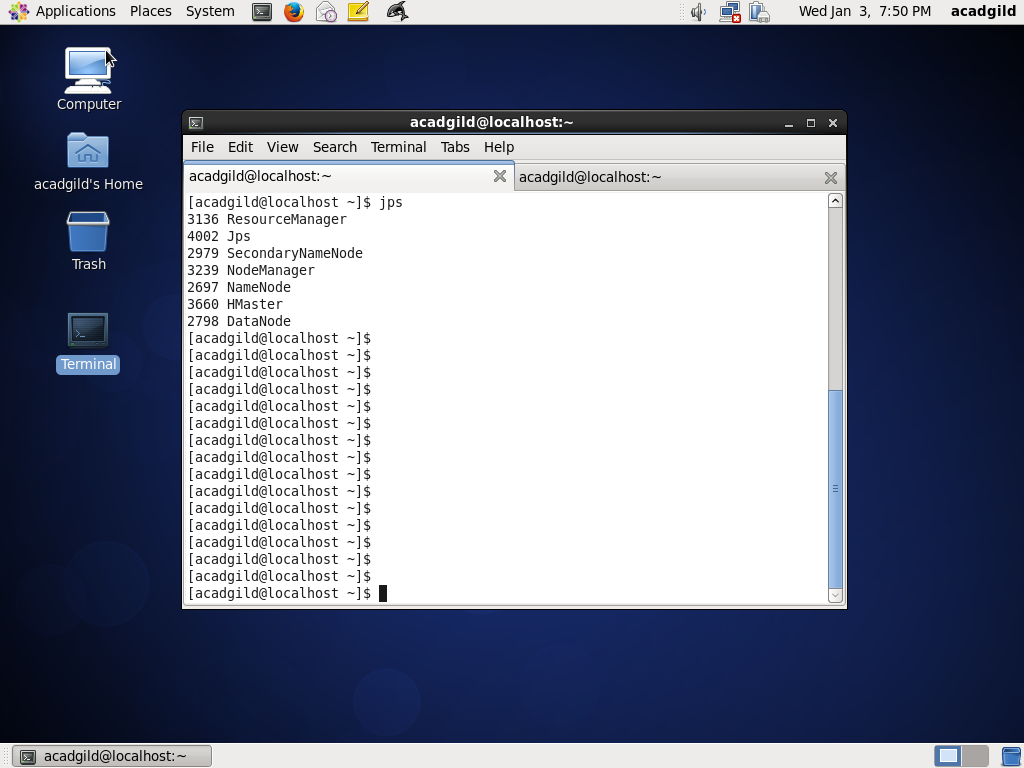
Start the services using start-all.sh

Start HBase using:

start-hbase.sh



Confirm that services are up using command: jps



Step2: Create class for loading Hbase tables

Create a project load-hbase-tables and create a scala class LoadHBaseTables which is explained below:

* Import all the dependent packages and create the package

package final\_project

import org.apache.log4j.{Level, LogManager, PropertyConfigurator}

import org.apache.spark.\_

import org.apache.spark.sql.SQLContext

import org.apache.hadoop.hbase.util.Bytes

import org.apache.hadoop.hbase.client.{ Put, HTable }

import org.apache.hadoop.hbase.HBaseConfiguration

import org.apache.hadoop.hbase.HTableDescriptor

import org.apache.hadoop.hbase.HColumnDescriptor

import org.apache.hadoop.hbase.client.HBaseAdmin

import org.apache.hadoop.hbase.client.HTable

import org.apache.spark.rdd.RDD

import org.apache.hadoop.hbase.mapreduce.TableInputFormat

import org.apache.log4j.{Level, LogManager, PropertyConfigurator}

* Create object LoadHBaseTables and define main method and logs

object LoadHBaseTables {

def main(args: Array[String]) {

val log = LogManager.getRootLogger

log.setLevel(Level.INFO)

* Get all the arguments, argument 1 is for filePath of lookup file, argument 2 is table name, argument 3 is composite fields of separated fields of Column Family and Column, which can be multiple

val filePath = args(0)

val tablename = args(1)

val columnFamilyField = args(2)

val columnFamilyFieldList = columnFamilyField.split(",")

val columnFamilyFieldListLength = columnFamilyFieldList.length

* Create configuration, Spark Context, SQL Context

val conf = new SparkConf().setAppName("LoadHbaseTable").setMaster("local[2]")

val sc = new SparkContext(conf)

sc.setLogLevel("ERROR")

val sqlContext = new SQLContext(sc)

* Create HBase configuration

val hconf = HBaseConfiguration.create

hconf.set(TableInputFormat.INPUT\_TABLE, tablename)

val admin = new HBaseAdmin(hconf)

* Check if tables is already created or not . If not created, create table and its column family

if (!admin.isTableAvailable(tablename)) {

log.info("Creating table " + tablename)

val tableDescription = new HTableDescriptor(tablename)

tableDescription.addFamily(new HColumnDescriptor(columnFamilyFieldList(0).getBytes()))

admin.createTable(tableDescription)

if (admin.isTableAvailable(tablename)) {

log.info("Table " + tablename + " is created successfully")

}

} else {

log.warn("Table " + tablename + " already exists")

}

* Get the table and

val table = new HTable(hconf, tablename)

* Process the csv file which has lookup data and create tuples using map with key and value. If there are multiple fields given, value is taken as a tuple

val file = sc.textFile("file://" + filePath)

var records1:Array[(String,String)] = null

var records2:Array[(String,String,String)] = null

if (columnFamilyFieldListLength == 2) {

records1 = file.map(\_.split(",")).map(x => (x(0), x(1))).collect

} else if (columnFamilyFieldListLength == 4) {

records2 = file.map(\_.split(",")).map(x => (x(0), x(1), x(2))).collect

}

* Save the columns

if (columnFamilyFieldListLength == 2) {

for(record <- records1) {

var p = new Put(new String(record.\_1).getBytes())

p.add(columnFamilyFieldList(0).getBytes(), columnFamilyFieldList(1).getBytes(), new String(record.\_2).getBytes())

table.put(p)

}

} else if (columnFamilyFieldListLength == 4) {

for(record <- records2) {

var p = new Put(new String(record.\_1).getBytes())

p.add(columnFamilyFieldList(0).getBytes(), columnFamilyFieldList(1).getBytes(), new String(record.\_2).getBytes())

table.put(p)

var q = new Put(new String(record.\_1).getBytes())

q.add(columnFamilyFieldList(2).getBytes(), columnFamilyFieldList(3).getBytes(), new String(record.\_3).getBytes())

table.put(q)

}

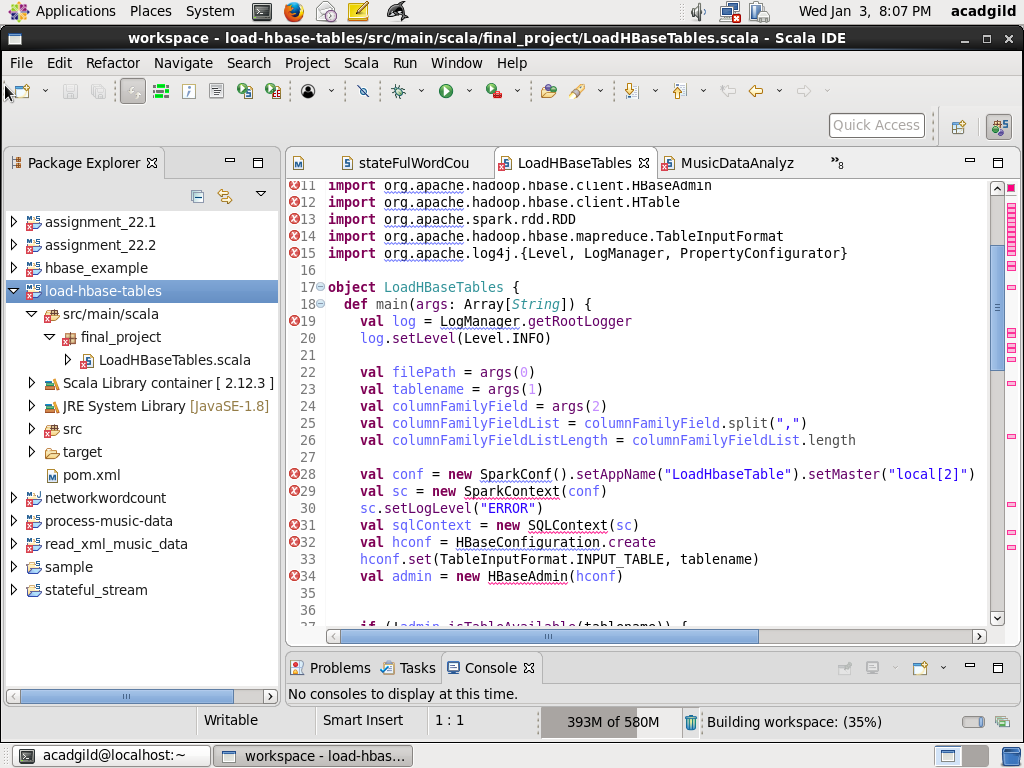
}

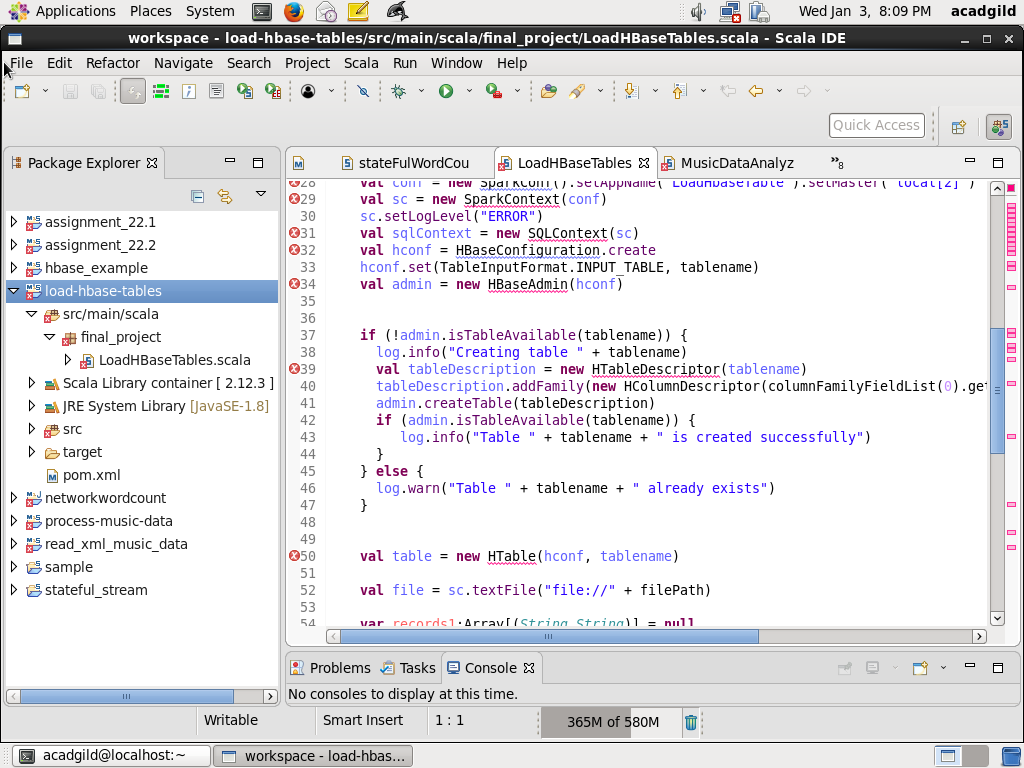
table.flushCommits()

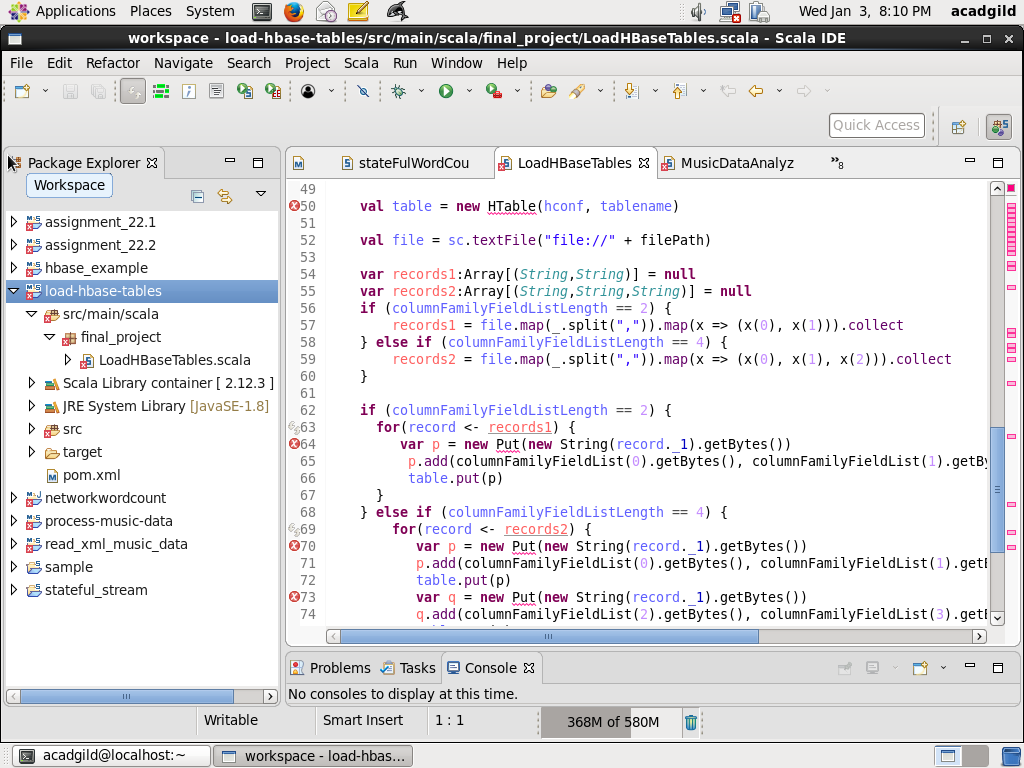
}

}

Screenshots are as below:

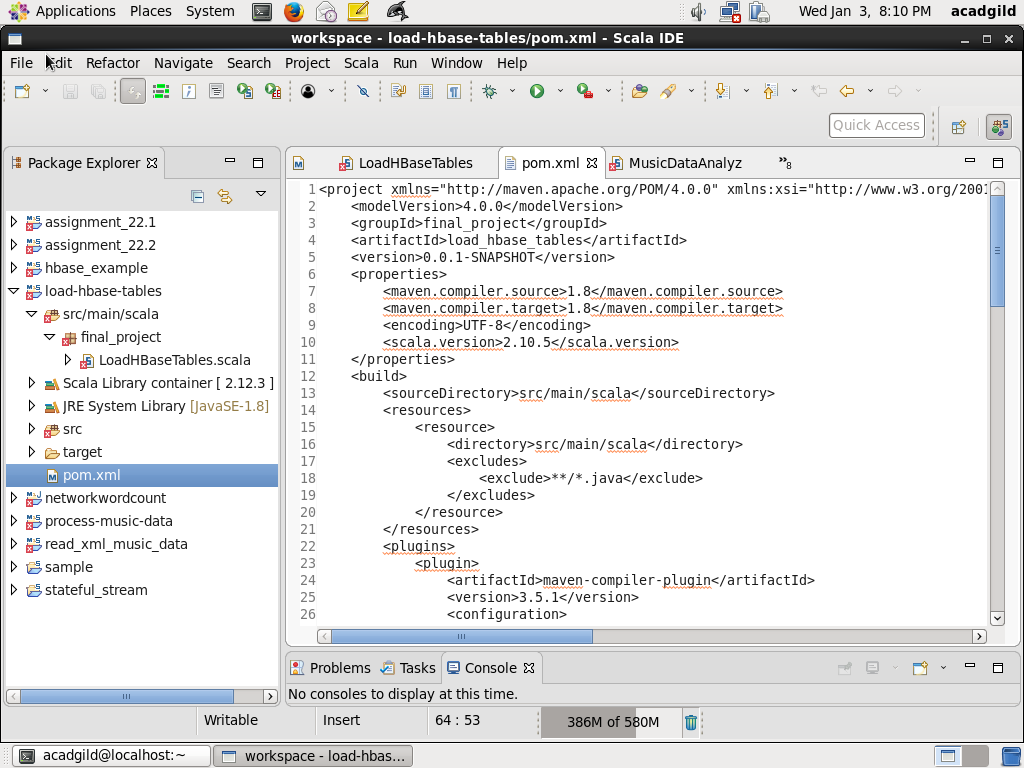


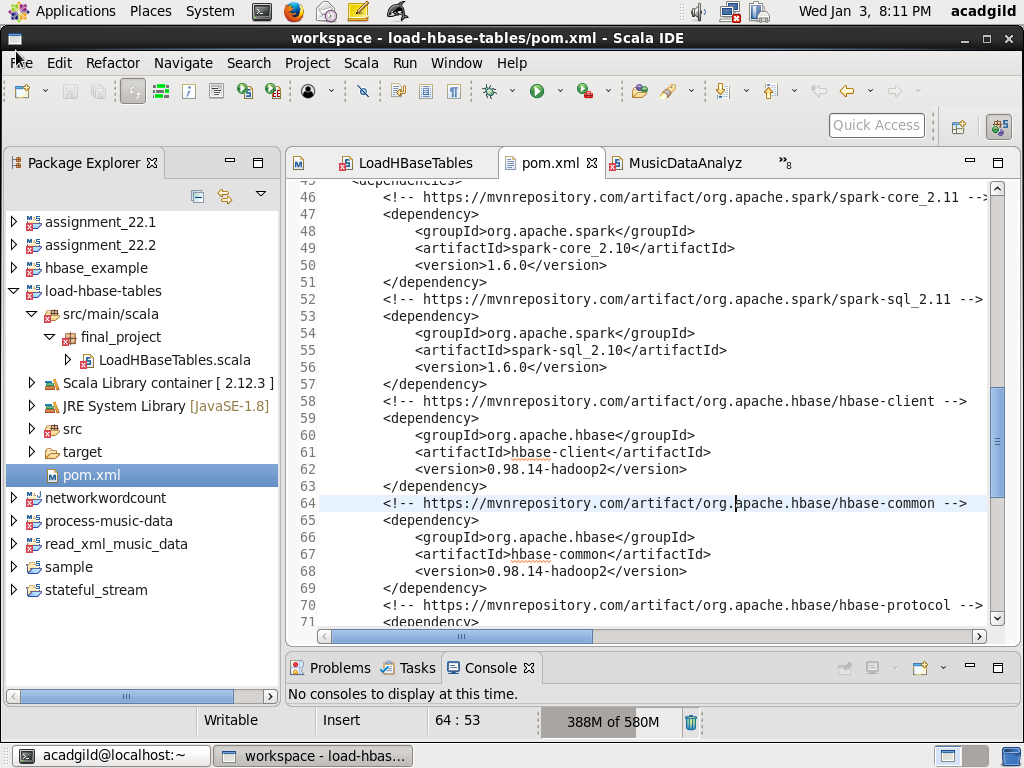




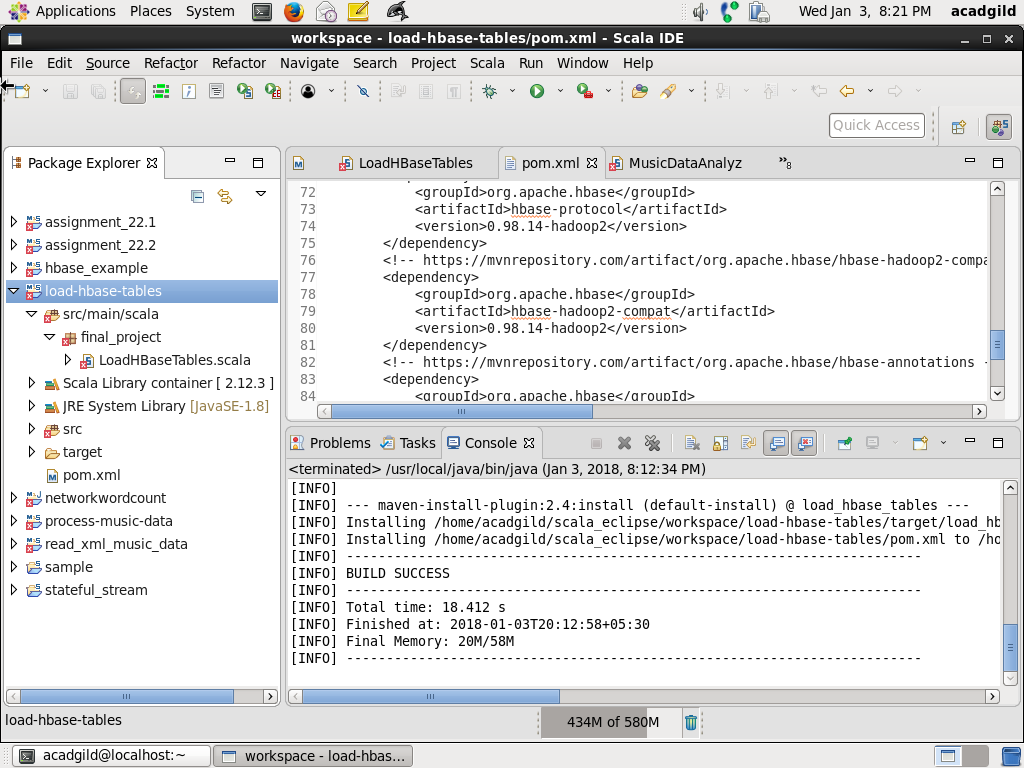
Step3: Compile the file using eclipse and run the code

* Define the pom file

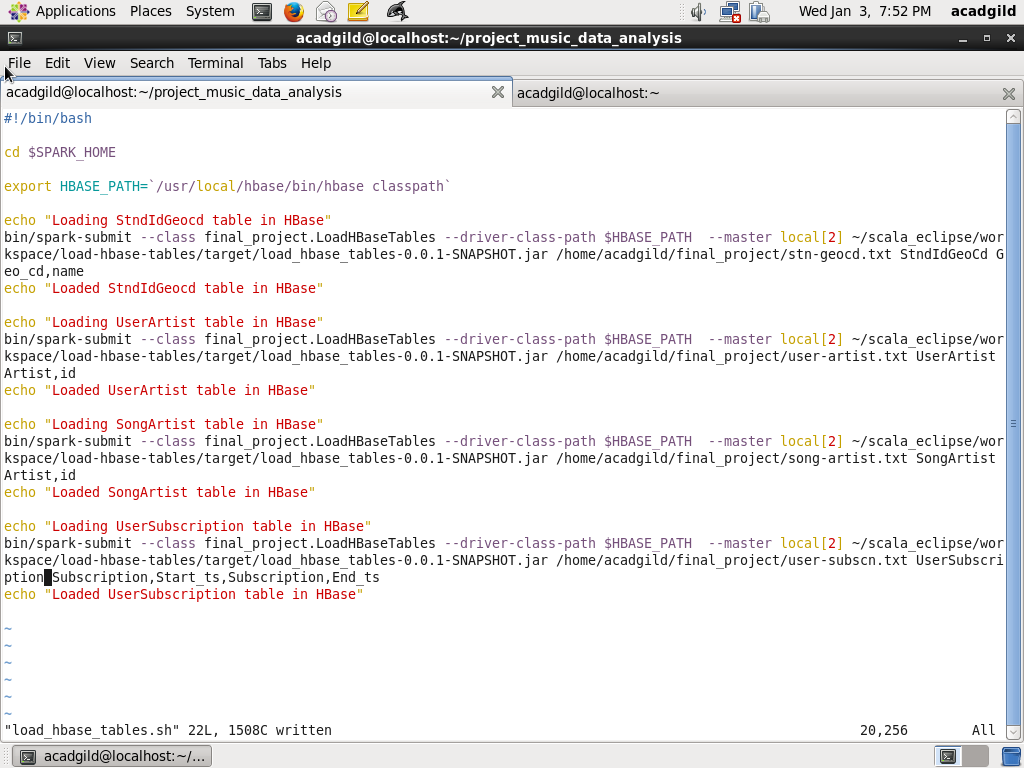




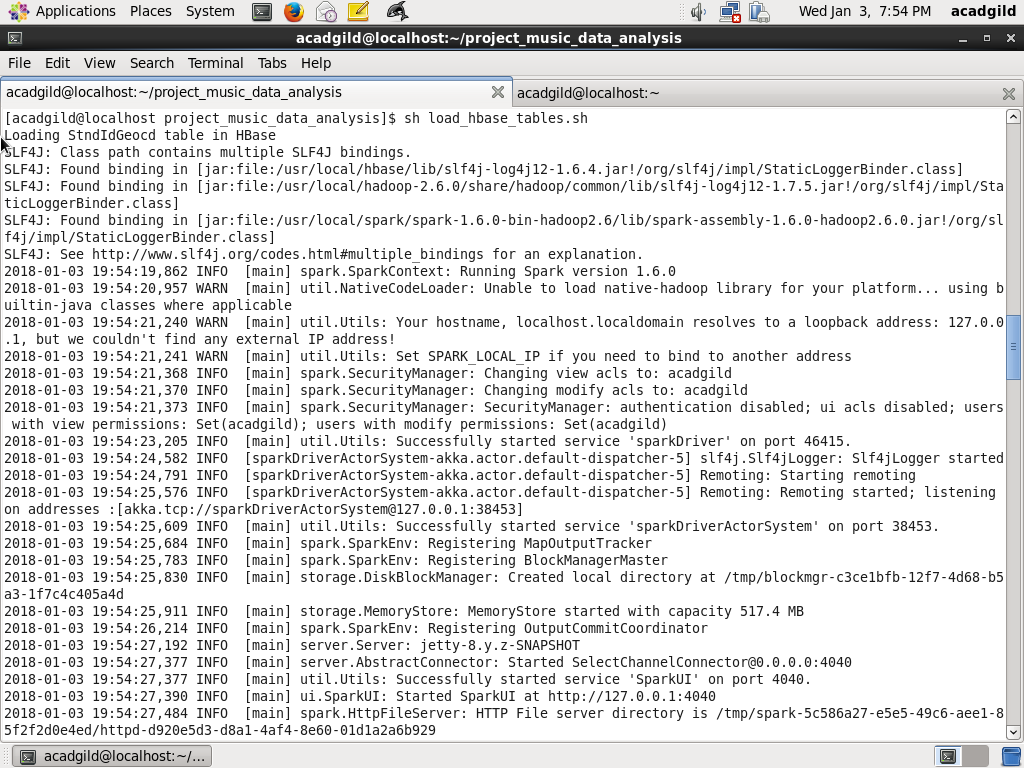
* Compile using maven install, screenshot is as below:

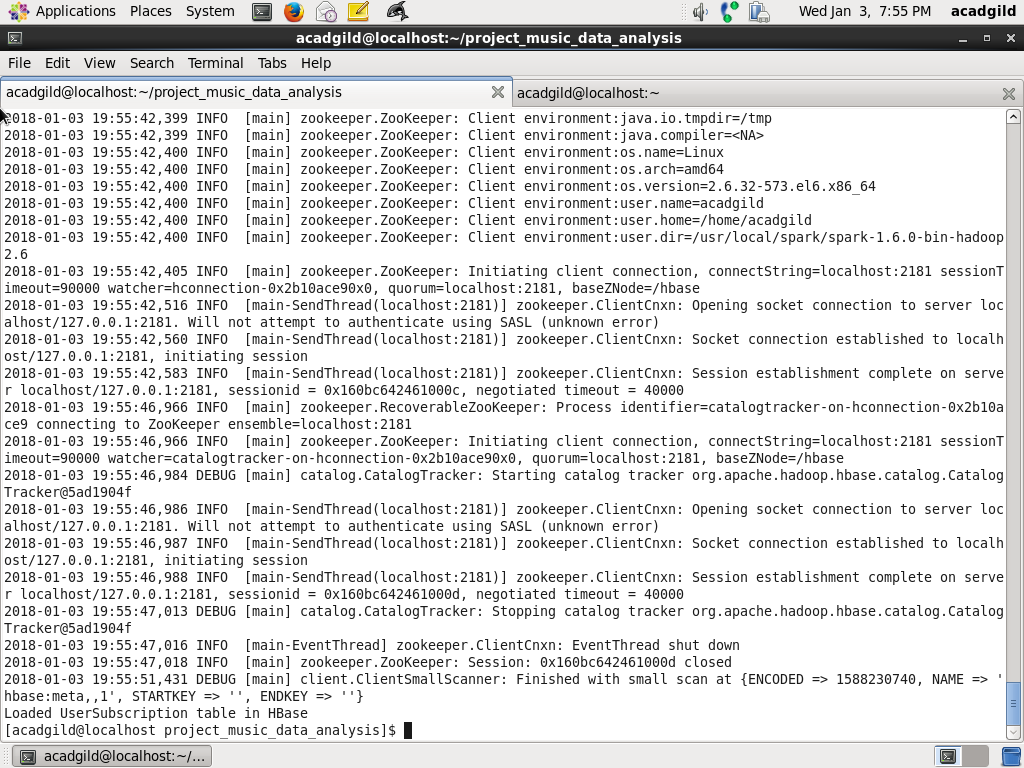


* Write a script to load all the tables:



* Run the script to load all the tables





Step3: Verify that all the lookup tables are populated correctly

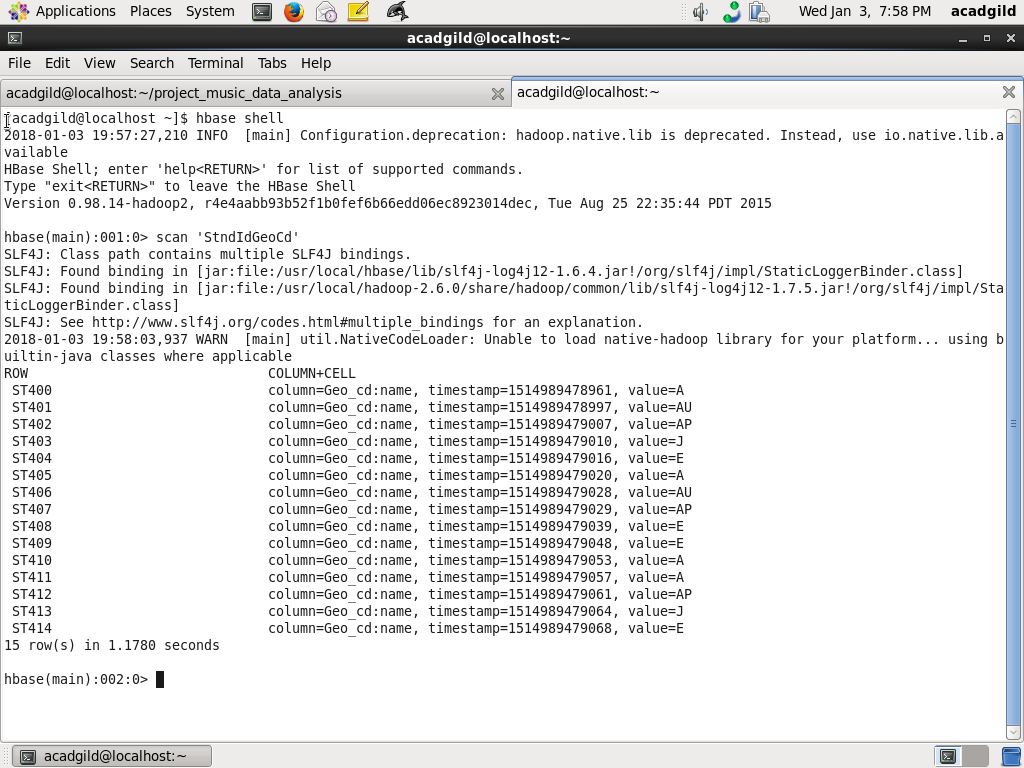
* Verify all the tables are populated correctly first login to HBase using

hbase shell

* Table StnIdGeoCd is verified using

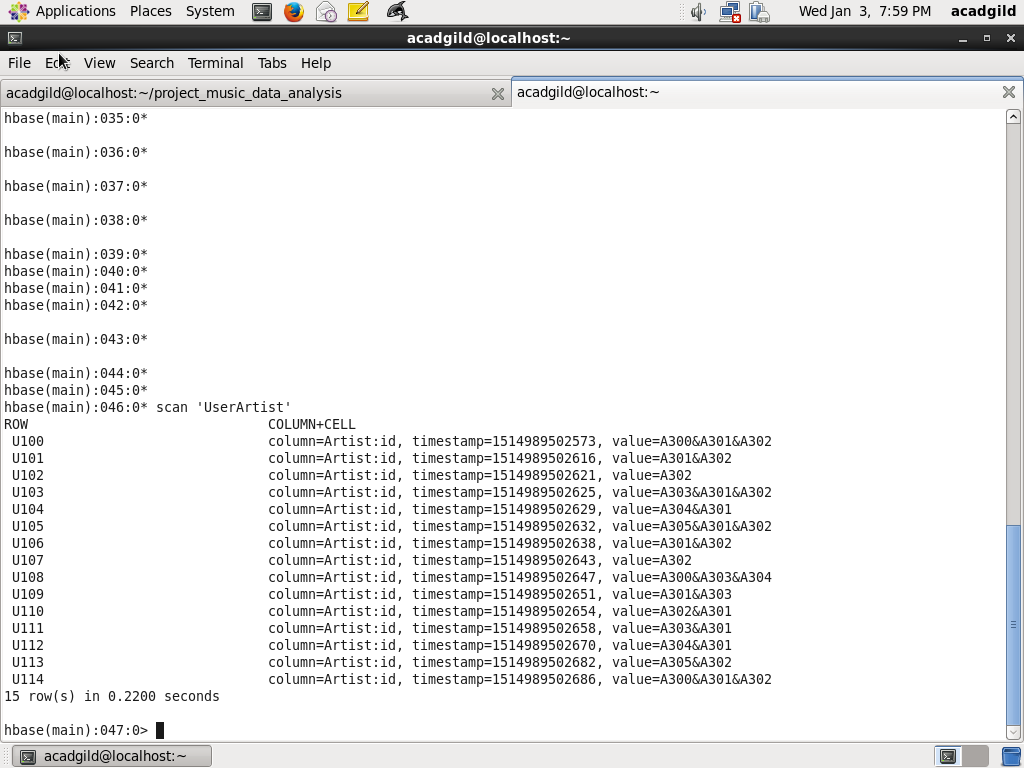
scan ‘StnIdGeoCd’

Screenshot is as below:

* 
* Table UserArtist is verified using

scan ‘UserArtist’

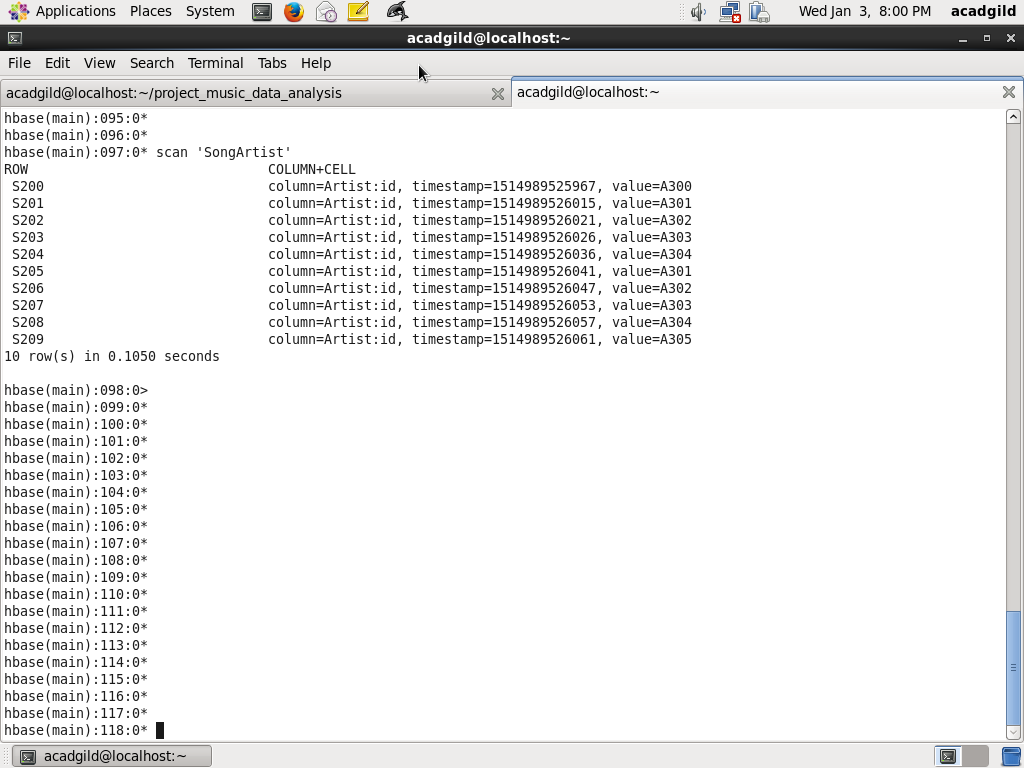
Screenshot is as below



* Table SongArtist is verified using

scan ‘SongArtist’

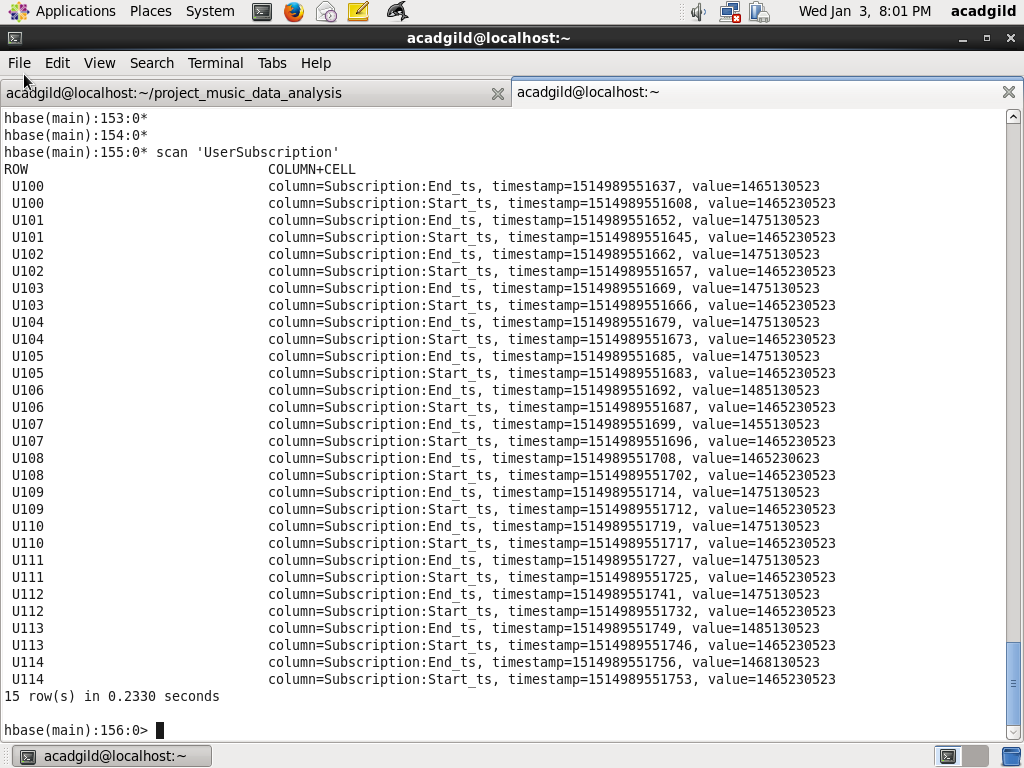
Screenshot is as below:



* Table UserSubscription is verified using

scan ‘UserSubscription’

Screenshot is as below:



**MODULE2: Handling Web and Mobile Data**

Step1: Write a scala class for handling Web data

* Write class WebMusicDataProcessor to process web music data stored in /data/web/file-1.xml and store it as dataframe

-Import dependent packages

package final\_project

import org.apache.spark.\_

import scala.xml.XML

import org.apache.spark.sql.DataFrame

import scala.collection.mutable.ListBuffer

import org.apache.spark.sql.SQLContext

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

import org.apache.log4j.{ Level, LogManager, PropertyConfigurator }

import org.apache.spark.broadcast.Broadcast

import java.text.SimpleDateFormat

import scala.collection.mutable.HashMap

// Define CustomException this is to solve the continue

case class CustomException(message:String) extends Exception(message)

// Define the class with WebMusicDataProcessor with parametes

class WebMusicDataProcessor(param: String, context: SparkContext, sqc: SQLContext) extends Serializable {

val filePath: String = param

val sc: SparkContext = context

val sqlContext:SQLContext = sqc

// Define the method which does the processing of data and return as dataframe

def processData(): DataFrame = {

val log = LogManager.getRootLogger

log.setLevel(Level.INFO)

// val sqlContext = new SQLContext(sc)

import sqlContext.implicits.\_

// Define dateFormat as yyyy-MM-dd HH:mm:ss and recordListBuffer which will act as buffer for storing data

val dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

val recordListBuffer = new ListBuffer[(String, String, String, Long, Long, Long, String, String, String, String, String)]()

// Load data from XML path

val xml = XML.loadFile(filePath)

* Process all the fields user\_id, song\_id, artist\_id, timestamp, start\_ts,end\_ts. Handle all null conditions

for (tag <- xml.child) {

val userId = (tag \ "user\_id").text

val songId = (tag \ "song\_id").text

var artistId = (tag \ "artist\_id").text

val timestamp = (tag \ "timestamp").text

var timestampLong:Long = 0

if (!timestamp.equals("")) {

timestampLong = dateFormat.parse(timestamp).getTime()

}

val startTs = (tag \ "start\_ts").text

var startTsLong:Long = 0

if (!startTs.equals("")) {

startTsLong = dateFormat.parse(startTs).getTime()

}

val endTs = (tag \ "end\_ts").text

var endTsLong:Long = 0

if (!endTs.equals("")) {

endTsLong = dateFormat.parse(endTs).getTime()

}

var geoCd = (tag \ "geo\_cd").text

val stationId = (tag \ "station\_id").text

val songEndType = (tag \ "song\_end\_type").text

var like = (tag \ "like").text

if (like.equals("")) like = "0"

var dislike = (tag \ "dislike").text

if (dislike.equals("")) dislike = "0"

try {

// Continue with the record in case fileds are blank

if (userId.equals("") && songId.equals("") && artistId.equals("")) {

throw CustomException("Record is blank")

} else {

recordListBuffer += ((userId, songId, artistId, timestampLong, startTsLong, endTsLong, geoCd, stationId, songEndType, like, dislike))

}

} catch {

case CustomException(msg) => msg

}

}

// From the buffer convert to dataFrame recordDF having fields User\_id, Songs\_id, Artist\_id, Timestamp, // Start\_ts, End\_ts, Geo\_cd, Station\_id, Song\_end\_type, Likes, Dislikes

val recordList = recordListBuffer.toList

val recordRDD = sc.parallelize(recordList)

val recordDF = recordRDD.toDF("User\_id", "Songs\_id", "Artist\_id", "Timestamp",

"Start\_ts", "End\_ts", "Geo\_cd", "Station\_id", "Song\_end\_type",

"Likes", "Dislikes")

log.info("Number of records =" + recordDF.count)

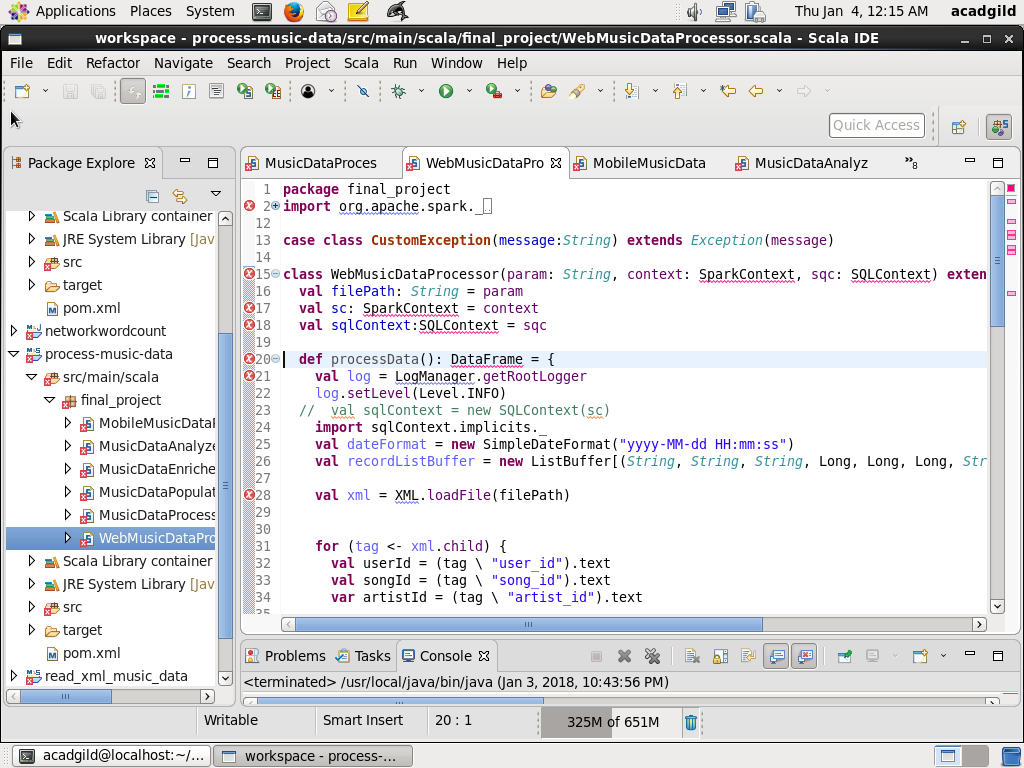
log.info("Showing records for Web Music Data ")

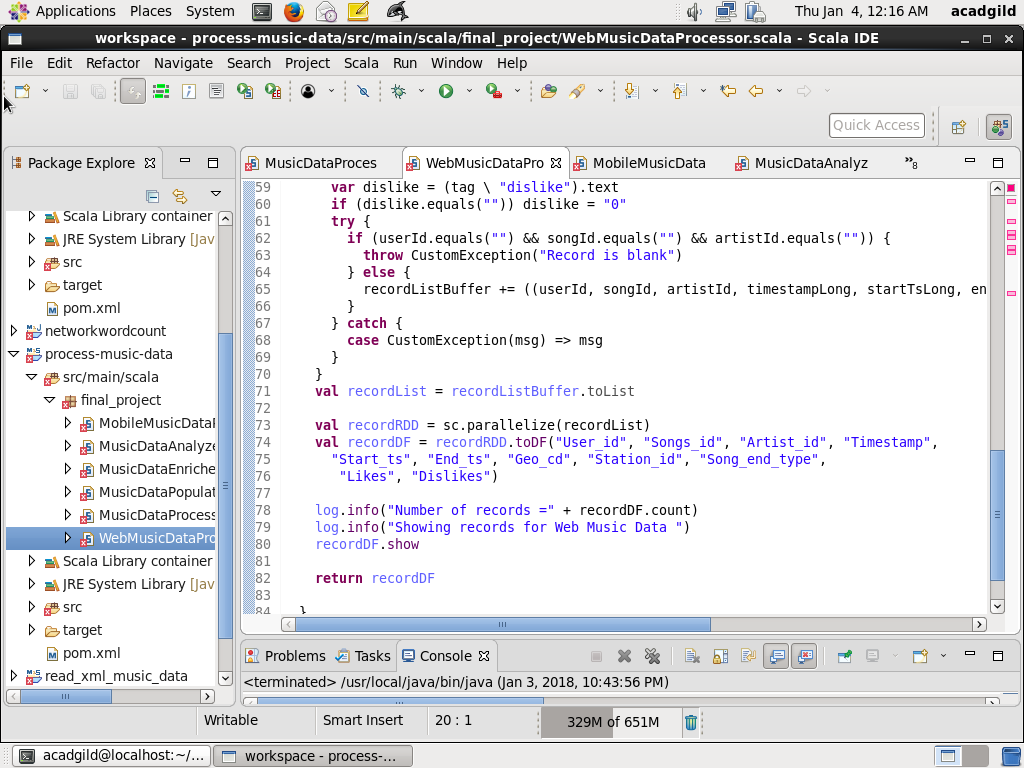
recordDF.show

return recordDF

}

Screenshot is as below:





Step2: define class for processing Mobile data

* Define a class MobileMusicDataProcessor for processing music data from /data/mob/file.txt

// define package as final\_project and Import al the dependent packages

package final\_project

import org.apache.spark.\_

import org.apache.spark.sql.DataFrame

import scala.collection.mutable.ListBuffer

import org.apache.spark.sql.SQLContext

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

import org.apache.log4j.{ Level, LogManager, PropertyConfigurator }

import org.apache.spark.broadcast.Broadcast

import scala.collection.mutable.HashMap

// Create a case class MusicData with all the fields

case class MusicData(User\_id:String, Songs\_id:String, Artist\_id:String, Timestamp:Long,

Start\_ts:Long, End\_ts:Long, Geo\_cd:String, Station\_id:String, Song\_end\_type:String,

Likes:String, Dislikes:String)

// Define class MobileMusicDataProcessor with all the parameters

class MobileMusicDataProcessor(param: String, context: SparkContext, sqc:SQLContext) extends Serializable {

val filePath: String = param

val sc: SparkContext = context

val sqlContext:SQLContext = sqc

// Define method processData

def processData(): DataFrame = {

val log = LogManager.getRootLogger

log.setLevel(Level.INFO)

import sqlContext.implicits.\_

// Load dataset from mobeile file path /data/mob/file.txt into RDD and slipt the fields

val recordRDD = sc.textFile(filePath)

val recordFieldsRDD = recordRDD.map(x => x.split(",")).filter(x=> x.length ==11)

// Convert the RDD to dataframe with case class MusicData

val recordDF = recordFieldsRDD.map(x => MusicData(x(0),

x(1),

x(2),

if (x(3).equals("")) 0 else x(3).toLong,

if (x(4).equals("")) 0 else x(4).toLong,

if (x(5).equals("")) 0 else x(5).toLong,

x(6),

x(7),

x(8),

if (x(9).equals("")) "0" else x(9),

if (x(10).equals("")) "0" else x(10))).toDF

log.info("Number of records for Mobile Music Data =" + recordDF.count)

log.info("Showing records for Mobile Music Data ")

recordDF.show

return recordDF

}

}

**MODULE3: Enrich and validate data**

Step1: Define a class MusicDataEnricher which does the enrichment and validation of fields in dataframe using UDF

// Define package final\_project and import all the dependent packages

package final\_project

import org.apache.spark.\_

import org.apache.spark.sql.DataFrame

import scala.collection.mutable.ListBuffer

import org.apache.spark.sql.SQLContext

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

import org.apache.log4j.{ Level, LogManager, PropertyConfigurator }

import org.apache.spark.broadcast.Broadcast

import org.apache.spark.sql.functions.udf

import scala.collection.mutable.HashMap

// Define class MusicDataEnricher with the parmeters of all SparkContext, DataFrame and all the

// broadcast maps which has lookup data

class MusicDataEnricher( allDataFrameParam: DataFrame, broadcastStndIdGeoCdMapParam:Broadcast[Map[String, String]], broadcastSongArtistMapParam:Broadcast[Map[String, String]], broadcastUserArtistMapParam: Broadcast[Map[String, String]], broadcastUserSubscriptionParam: Broadcast[Map[String, (Long, Long)]]) extends Serializable {

val allDataFrame: DataFrame =allDataFrameParam

// val stndIdGeoCdMap:HashMap[String, String] = stndIdGeoCdMapParam

val broadcastStndIdGeoCdMap:Broadcast[Map[String, String]] = broadcastStndIdGeoCdMapParam

val broadcastSongArtistMap:Broadcast[Map[String, String]] = broadcastSongArtistMapParam

val broadcastUserArtistMap:Broadcast[Map[String, String]] = broadcastUserArtistMapParam

// The UDF method fillNullValueGeoCd wlll take stationId and geoCd as parameter and if geoCd is not

// blank then it will take as it is. If geoCd is blank it will use the lookup map broadcastStndIdGeoCdMap

// using stationId, get the geoCd. If it is not there record will be marked as Invalid

// A new field modified\_Geo\_cd will be added to the dataframe

def fillNullValueGeoCd = udf((stationId: String, geoCd: String) => {

if (!geoCd.equals("")) geoCd

else {

// val geoCdVal = stndIdGeoCdMap.get(stationId).getOrElse("Invalid")

val geoCdVal = broadcastStndIdGeoCdMap.value.get(stationId).getOrElse("Invalid")

geoCdVal

}

})

// The UDF method fillNullValueArtistId wlll take songId and artistId as parameter and if artistId is not

// blank then it will take as it is. If artistId is blank it will use the lookup map broadcastSongArtistMap

// using songId, get the artistId. If it is not there record will be marked as Invalid

// A new field modified\_Artist\_id will be added to the dataframe

def fillNullValueArtistId = udf((songId: String, artistId: String) => {

if (!artistId.equals("")) artistId

else {

val artistIdVal = broadcastSongArtistMap.value.get(songId).getOrElse("Invalid")

artistIdVal

}

})

// The UDF method findFollowers wlll take userId and artistId as parameter.

// Based on usedId from the map broadcastUserArtistMap, artistList is retrieved. If aritstId list is blank

// then 0 will be retuned. If artistList is not blank then it will be split based on & and create a array

// artistArray. If it contains using artistId, 1 will be returned, else 0 will be returned

// A new field follower will be added to the dataframe

def findFollowers = udf((userId: String, artistId: String) => {

val artistList = broadcastUserArtistMap.value.get(userId).getOrElse("")

if (artistList.equals("")) "0"

else {

var artistArray = artistList.split("&")

if (artistArray contains artistId) "1"

else "0"

}

})

// The UDF method findSubscribers wlll take userId and starts as parameter. Basedon userId

// lokkup is done on broadcastUserSubscription.value.get(userId). If subscription does not exits, it

// returns 0. If it exists and starts is in between subscription start time and end time return 1 else

/ /retun 0. A new field subscribed is added to the dataframe

def findSubscribers = udf((userId: String, startTs: Long) => {

val subscriptionTuple= broadcastUserSubscription.value.get(userId).getOrElse((0L, 0L))

if (subscriptionTuple.\_1 ==0 && subscriptionTuple.\_2 == 0 ) "0"

else if (startTs >= subscriptionTuple.\_1 && startTs <= subscriptionTuple.\_2) "1"

else "0"

})

// Using UDF validate records validation is done. If userId, songId are blank return 0. If modifiedArtistId // or modifiedGeoCdi si Invalid return 0,. If timestamp or start\_ts is 0 then 0. If end\_ts is less than // start\_ts return 0. Else return 1. Add a new field isValid to the dataframe

def validateRecords = udf((userId: String, songId: String, modifiedArtistId: String, modifiedGeoCd: String, timestamp:Long, start\_ts:Long, end\_ts:Long) => {

if (userId.equals("")) "0"

else if (songId.equals("")) "0"

else if (modifiedArtistId.equals("Invalid")) "0"

else if (modifiedGeoCd.equals("Invalid")) "0"

else if (timestamp == 0) "0"

else if(start\_ts == 0) "0"

else if (end\_ts < start\_ts) "0"

else "1"

})

// The method enrichData will execute all the UDFs defined above and enrich dataframe with new fields

def enrichData():DataFrame = {

var newDataFrame = allDataFrame.withColumn("modified\_Geo\_cd", fillNullValueGeoCd(allDataFrame("Station\_id"), allDataFrame("Geo\_cd")))

newDataFrame = newDataFrame.withColumn("modified\_Artist\_id", fillNullValueArtistId(newDataFrame("Songs\_id"), newDataFrame("Artist\_id")))

newDataFrame = newDataFrame.withColumn("follower", findFollowers(newDataFrame("User\_id"), newDataFrame("modified\_Artist\_id")))

newDataFrame = newDataFrame.withColumn("subscribed", findSubscribers(newDataFrame("User\_id"), newDataFrame("Start\_ts")))

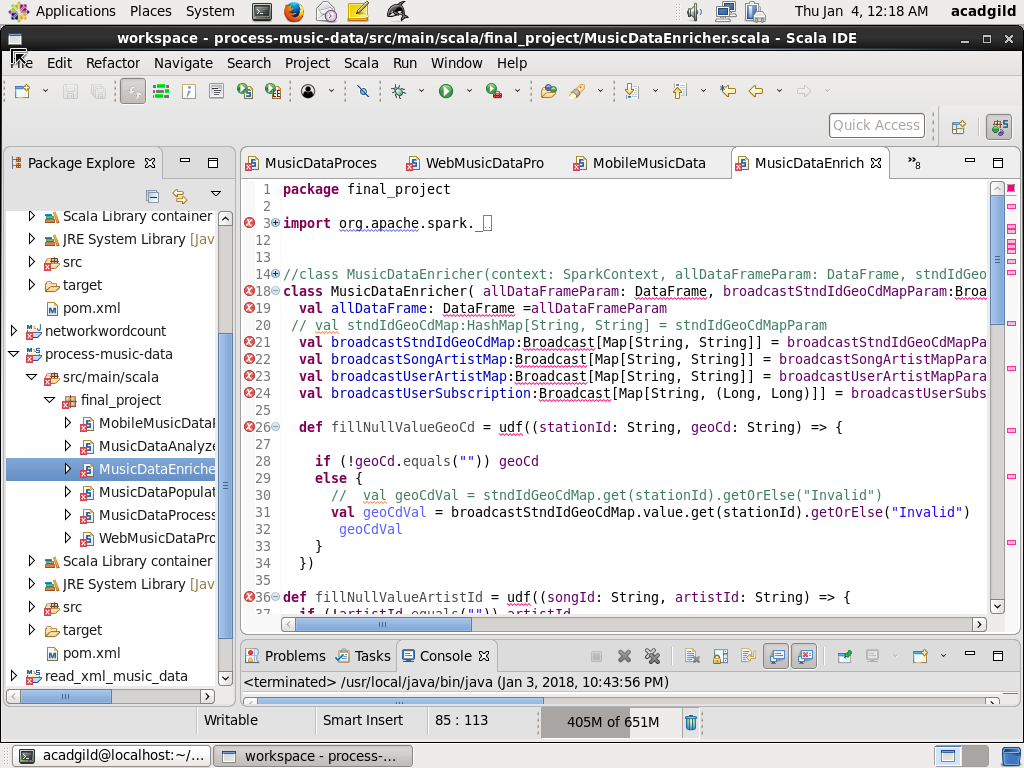
newDataFrame = newDataFrame.withColumn("isValid", validateRecords (newDataFrame("User\_id"), newDataFrame("Songs\_id"), newDataFrame("modified\_Artist\_id"), newDataFrame("modified\_Geo\_cd"), newDataFrame("Timestamp"), newDataFrame("Start\_ts"), newDataFrame("End\_ts")))

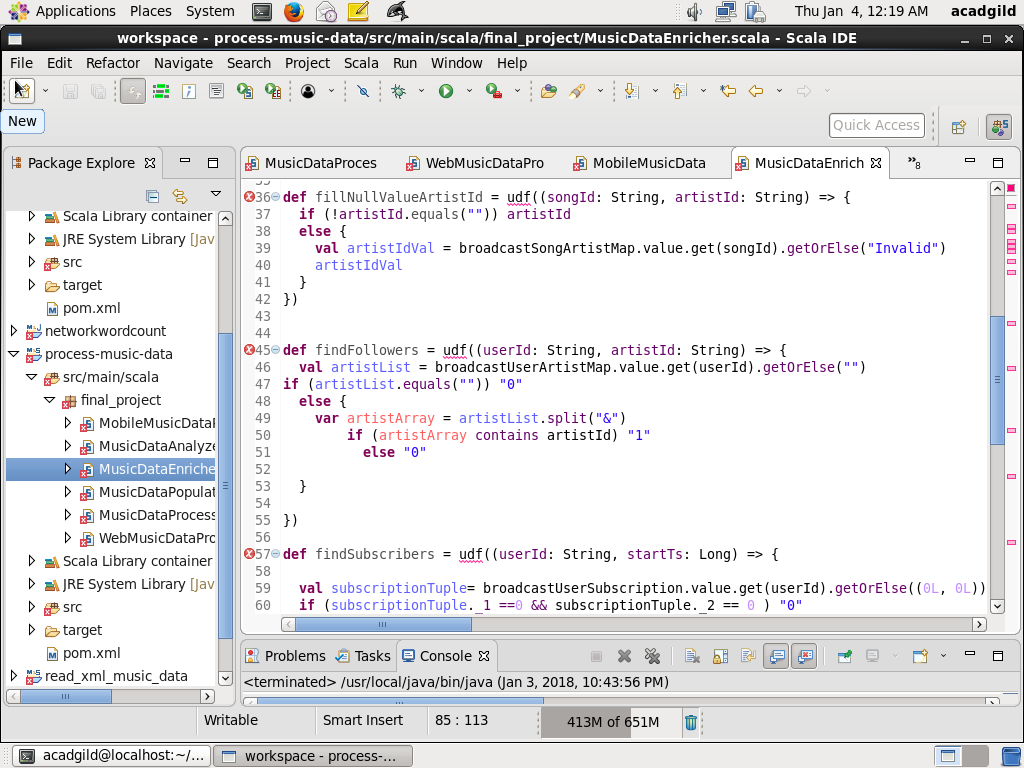
return newDataFrame

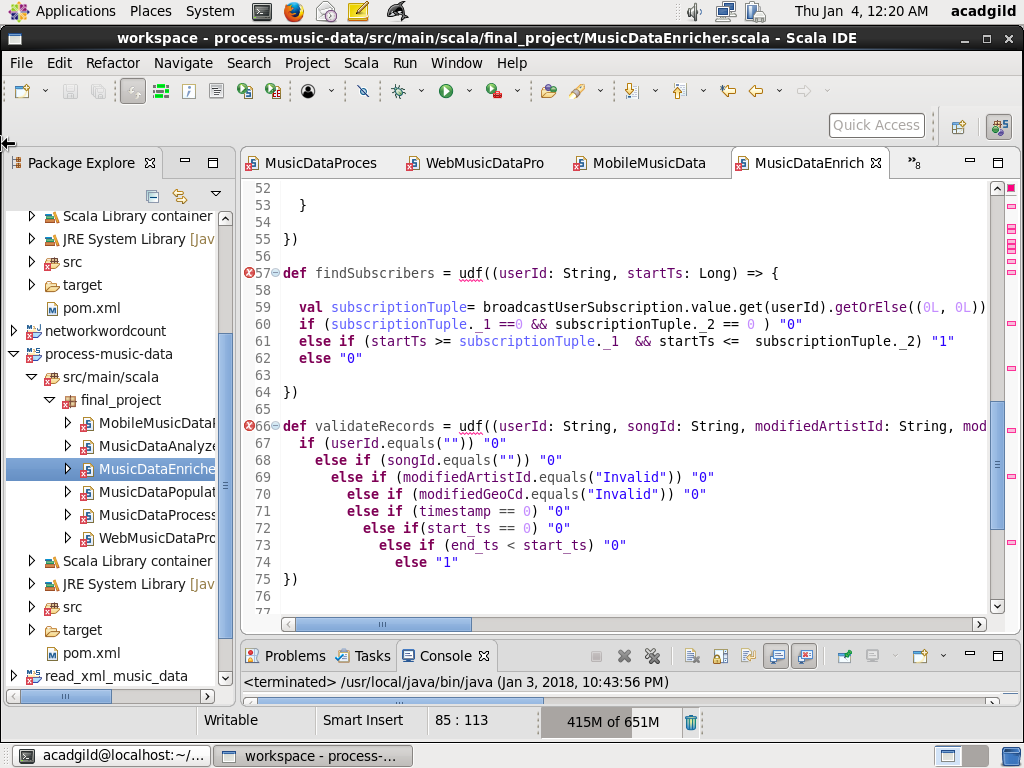
}

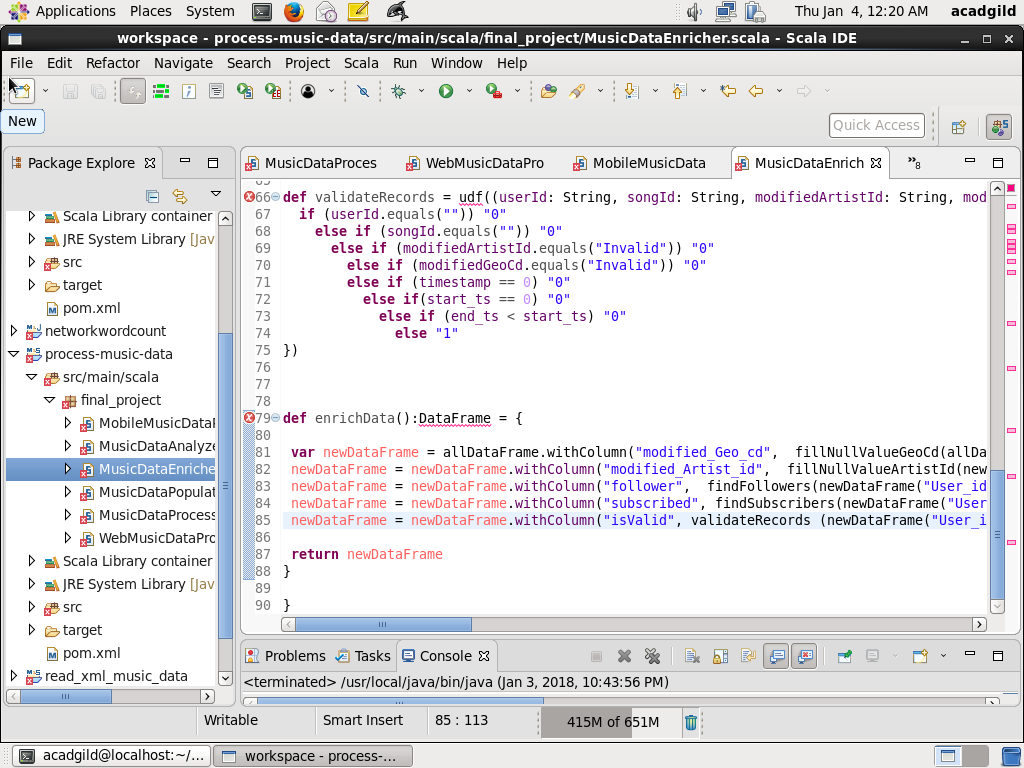
}

Screenshot is as below:









**MODULE4: Analyze the data**

* Analyze the data using defining MusicDataAnalyzer class, which will execute sql queries and the result will be stored as reports in HDFS

// Define the package final\_project and import all the dependent packages

package final\_project

import org.apache.log4j.{ Level, LogManager, PropertyConfigurator }

import org.apache.spark.sql.AnalysisException

import org.apache.spark.\_

import org.apache.spark.sql.SQLContext

import org.apache.spark.sql.DataFrame

import scala.collection.mutable.HashMap

// Define the class MusicDataAnalyzer with parameters SparkContext, SQLContext and DataFrame

class MusicDataAnalyzer(context: SparkContext, sqc: SQLContext, musicDataDFParam: DataFrame) extends Serializable {

val musicDataDF: DataFrame = musicDataDFParam

val sc = context

val sqlContext = sqc

// Define logger and reportBasePath and currentTimestamp

val log = LogManager.getRootLogger

log.setLevel(Level.INFO)

val reportBasePath = "/user/acadgild/project\_music\_data\_analysis/reports/"

val currentTimestamp = System.currentTimeMillis().toString

// Define method analyze which will in run call all the methods for analysis

// While calling method, each one is put in try catch block so that failing one does not impact others

def analyze() = {

import sqlContext.implicits.\_

// Define temporary table MusicDataDetailed on the dataframe

musicDataDF.registerTempTable("MusicDataDetailed")

log.info("Before calling getAllRecords()")

try {

getAllRecords()

} catch {

case e: Exception => log.error("Exception got while calling getAllRecords: " + e)

}

log.info("Before calling getTop10Stations()")

try {

getTop10Stations()

} catch {

case e: Exception => log.error("Exception got while calling getTop10Stations: " + e)

}

log.info("Before calling getMusicDurtionByUser()")

try {

getMusicDurtionByUserType()

} catch {

case e: Exception => log.error("Exception got while calling getTop10Stations: " + e)

}

log.info("Before calling getTo10ConnectedArtists()")

try {

getTop10ConnectedArtists()

} catch {

case e: Exception => log.error("Exception got while calling getTop10ConnectedArtists: " + e)

}

log.info("Before calling getTop10UnsubscribedUsers()")

try {

getTop10UnsubscribedUsers()

} catch {

case e: Exception => log.error("Exception got while calling getTop10UnsubscribedUsers(): " + e)

}

log.info("Before calling getTo10SongsHavignMaximumRevenue()")

try {

getTop10SongsHavingMaximumRevenue()

} catch {

case e: Exception => log.error("Exception got while calling getTop10SongsHavingMaximumRevenue(): " + e)

}

}

// Method getAllRecords will select all the records from MusicDataDetailed

def getAllRecords() {

log.info(" Get All records")

// Run the query to get the result

val df = sqlContext.sql("SELECT \* FROM MusicDataDetailed ")

df.show()

// Store the result as a single file in HDFS with reportPath and report name MusicDataAllRecords concatenated with current timestamp

val df1 = df.repartition(1)

df1.write.format("com.databricks.spark.csv").option("header", "true").save(reportBasePath + "/MusicDataAllRecords\_" + currentTimestamp)

}

// The method getTop10Stations will return top 10 stations were maximum songs are played which are

// liked by unique user

def getTop10Stations() {

log.info(" Top 10 Music Stations where maximum numbers of songs played which are liked by unique users")

// Execute query to get Station\_id User\_id and count of music played Group by Station\_id, User\_id and // Likes is 1 and isValid is 1

sqlContext.sql("SELECT Station\_id, User\_id, count(\*) AS music\_count FROM MusicDataDetailed "

+ " WHERE Likes='1' AND isValid='1' GROUP BY Station\_id, User\_id")

.registerTempTable("MusicCountByStation")

// Execute query to get unique unique count for user liked, so music\_count is greater than 1, it is

// considered 1

sqlContext.sql("SELECT Station\_id, User\_id, CASE WHEN music\_count> 1 THEN 1 ELSE music\_count END "

+ " AS unique\_music\_count FROM MusicCountByStation")

.registerTempTable("UniqueMusicCountByStation")

// Using sum method of SQL, aggregate the total music count group by Station\_id and order

// total\_music\_count desceding. Take first 10 records

val df = sqlContext.sql("SELECT Station\_id, sum(unique\_music\_count) AS total\_music\_count "

+ " FROM UniqueMusicCountByStation GROUP BY Station\_id ORDER BY total\_music\_count "

+ " DESC LIMIT 10 ")

df.show()

// Store the query output to HDFS as a csv report Top10Stations concatenated with timestamp in

// reportBasePath

val df1 = df.repartition(1)

df1.write.format("com.databricks.spark.csv").option("header", "true").save(reportBasePath + "/Top10Stations\_" + currentTimestamp)

}

// The method getMusicDurtionByUserType returns total lke music by each category of user Subscribed

// or Unsubscribed

def getMusicDurtionByUserType() {

log.info(" Total duration of Songs played by Subscibed and Unsubsribed Users")

sqlContext.sql("SELECT CASE WHEN subscribed='1' THEN 'Subscribed' ELSE 'Unsubscribed' END AS User\_type, (End\_ts -Start\_ts) AS duration "

+ " FROM MusicDataDetailed WHERE isValid='1'")

.registerTempTable("UserTypeDuration")

val df = sqlContext.sql("SELECT User\_type, SUM(duration) AS total\_duration\_milliseconds FROM UserTypeDuration "

+ " GROUP BY User\_type ORDER BY total\_duration\_milliseconds DESC")

df.show()

val df1 = df.repartition(1)

df1.write.format("com.databricks.spark.csv").option("header", "true").save(reportBasePath + "/MusicDurationByUserType\_" + currentTimestamp)

}

// The method getTop10ConnectedArtists return top 10 connected artists who are followed by user

def getTop10ConnectedArtists() {

log.info(" Top 10 Connected Artists")

sqlContext.sql("SELECT Artist\_id, User\_id, count(\*) AS music\_count FROM MusicDataDetailed "

+ " WHERE follower='1' AND isValid='1' GROUP BY Artist\_id, User\_id")

.registerTempTable("MusicCountByArtist")

sqlContext.sql("SELECT Artist\_id, User\_id, CASE WHEN music\_count> 1 THEN 1 ELSE music\_count END "

+ " AS unique\_music\_count FROM MusicCountByArtist")

.registerTempTable("UniqueMusicCountByUser")

val df = sqlContext.sql("SELECT Artist\_id, sum(unique\_music\_count) AS total\_music\_count "

+ " FROM UniqueMusicCountByUser GROUP BY Artist\_id ORDER BY total\_music\_count "

+ " DESC LIMIT 10 ")

df.show()

val df1 = df.repartition(1)

df1.write.format("com.databricks.spark.csv").option("header", "true").save(reportBasePath + "/Top10ConnectedArtists\_" + currentTimestamp)

}

// The method getTop10SongsHavingMaximumRevenue returns top 10 songs having maximum royalty r

//revenue

def getTop10SongsHavingMaximumRevenue() {

log.info(" Top 10 Songs Having maximum revenue")

sqlContext.sql("SELECT Songs\_id, CASE WHEN End\_ts is NOT NULL AND Start\_ts is NOT NULL and End\_ts > Start\_ts THEN End\_ts - Start\_ts ELSE 0 END AS duration FROM MusicDataDetailed "

+ " WHERE (Likes='1' OR Song\_end\_type = '0') AND isValid='1' ")

.registerTempTable("SongDuration")

val df = sqlContext.sql("SELECT Songs\_id, SUM(duration) AS total\_duration\_milliseconds FROM SongDuration "

+ " GROUP BY Songs\_id ORDER BY total\_duration\_milliseconds DESC LIMIT 10")

df.show()

val df1 = df.repartition(1)

df1.write.format("com.databricks.spark.csv").option("header", "true").save(reportBasePath + "/Top10SongsHavignMaximumRevenue\_" + currentTimestamp)

}

// The method getTop10UnsubscribedUsers will retrun top 10 unsubscribed users

def getTop10UnsubscribedUsers() {

log.info(" Top 10 Unsubscribed Users")

sqlContext.sql("SELECT User\_id,CASE WHEN End\_ts is NOT NULL AND Start\_ts is NOT NULL and End\_ts > Start\_ts THEN End\_ts - Start\_ts ELSE 0 END AS duration FROM MusicDataDetailed "

+ " WHERE subscribed='0' AND isValid='1'")

.registerTempTable("SongDuration")

val df = sqlContext.sql("SELECT User\_id, SUM(duration) AS total\_duration\_milliseconds FROM SongDuration "

+ " GROUP BY User\_id ORDER BY total\_duration\_milliseconds DESC LIMIT 10")

df.show()

val df1 = df.repartition(1)

df1.write.format("com.databricks.spark.csv").option("header", "true").save(reportBasePath + "/Top10UnsubscribedUsers\_" + currentTimestamp)

}

}