quantexa
outcomes delivered

Coding Assignment

Introduction

The aim of this exercise is to test your programming ability.

The solution should be written in Scala or Spark/Scala, as this is the primary language used at Quantexa, and you should aim to write the solution in a functional style. If you find it helpful you may use any functionality from the standard library in your solution, but please avoid using other libraries.

Please note we will be reviewing both the functionality of your code as well as the style so please ensure you read the Scala coding guidelines, found in the Technical Best Practice, section of the training, before attempting your answers.

Assignment

A common problem we face at Quantexa is having lots of disjointed raw sources of data, from which relevant pieces of information must be aggregated and collected into hierarchical case classes, which we refer to as Documents. Documents represent a hypothetical global data source, which contains all the information that is needed for a use case. This exercise simplifies the realities of this, with just two initial sources of high quality data, however reflects the types of transformations that must be performed in a real-world scenario. Given customer information and account information separately, the objective is to aggregate this information and match customers to their respective accounts.

Data

You have been provided with customer data and account data that contain the information described below.

Customer Data:

Field	**Description**
customerId	A unique ID for each individual customer in the data.
forename	A customer's first name.
surname	A customer's last name.

Account Data:

Field	**Description**
customerId	A unique ID for each individual customer in the data.
accountId	A unique ID for each individual account in the data.
balance	An account's current balance($).

Capital Tower, 91 Waterloo Road, SE1 8RT
+44 203 808 8299 | www.quantexa.com

Output

The expected output which sufficiently answers the assigned questions is described in the following tables. These contain both the final output format and any accompanying data structures required (e.g. case classes).

Please note that your results should adhere to the naming and Type conventions specified in the below tables. This allows for a better and more efficient evaluation of your solution on our behalf.

Type	Name	Description
Dataset	customerAccountOutputDS	A Dataset, which contains elements of the CustomerAccountOutput case class.

Table 1: Expected output format

Type	Name	Fields	Field Type	Description
Case class	CustomerAccountOutput	customerId forename surname accounts numberAccounts totalBalance averageBalance	String String String Seq[AccountData] BigInt Long Double	A case class containing the relevant output information.

Table 2: Auxiliary case class

Code

The following code imports the data as Datasets and prints the first few records to the console. This allows you to use the Dataset API which includes .map/.groupByKey/.joinWith etc.

A Dataset also includes all of the DataFrame functionality, allowing you to use .join ("left-outer","inner" etc.). The challenge here is to group, aggregate, join and map the customer and account data given into the hierarchical case class “CustomerAccountOutput”. For this exercise we would like you to avoid using functions available in `import spark.sql.functions._`, should you choose to perform Dataframe transformations. We also prefer the use of the Datasets API.

(See here for more info: https://spark.apache.org/docs/latest/sql-programming-guide.html)

Scala 2.11:

```
import org.apache.log4j.{Level, Logger}
import org.apache.spark.sql.{Dataset, SparkSession}

object AccountsAssignment {

  //Create a spark context, using a local master so Spark runs on the local
machine
```

```
  val spark =
SparkSession.builder().master("local[*]").appName("AccountAssignment").getOrCreat
e()

  //importing spark implicits allows functions such as dataframe.as[T]
  import spark.implicits._

  //Set logger level to Warn
  Logger.getRootLogger.setLevel(Level.WARN)

  //Get file paths from files stored in project resources
  val customerCSV = getClass.getResource("/customer_data.csv").getPath
  val accountCSV = getClass.getResource("/account_data.csv").getPath


  //Create DataFrames of sources
  val customerDF = spark.read.option("header","true")
    .csv(customerCSV)
  val accountDF = spark.read.option("header","true")
    .csv(accountCSV)

  case class CustomerData(customerId: String, forename: String, surname: String)

  case class AccountData(customerId: String, accountId: String, balance: Long)



  //Create Datasets of sources
  val customerDS = customerDF.as[CustomerData]
  val accountDS =
accountDF.withColumn("balance",'balance.cast("long")).as[AccountData]

  customerDS.show
  accountDS.show

  //END GIVEN CODE
}
```