Coding Assessment

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 below before attempting your answers.

Remember this assessment is designed to cover the concepts and constructs that will be used in delivering a

Quantexa Platform solution. We will be assessing you on the following:

- Correct answer for all 3 questions
- Follows basic best practice code guidelines
- Neat and concise code
- No mutability of variables
- Use of pattern mattching, mapgroups, and groupby
- Handling of None/Option values
- Scala/Spark implementation

You have been provided with an example in which basic concepts are demonstrated

Quantexa Coding Guidelines:

General Scala/Spark	
Camel Case	Use camel casing, not underscores! Vals/vars/methods(defs) should all be lowerCamelCase, types/objects should all be UpperCamelCase (also known as ProperCase), for further details please see this style guide. **Exceptions** 1. In ETL processes, the raw data should be read into a parquet maintaining all of the source field names exactly. 2. Constant values - use ProperCase. For example, val YearInDays = 365 3. When using acronyms and camel case, if it is the first word within a variable name it should be lower case, if it is not the first word, it should be all upper case. E.G: • jiraTaskDescription (and not JIRATaskDescription) • accountIBAN (and not accountIban) 4. When using acronyms and proper case, every letter of the acronym should be upper case, e.g. • case class ETLConfig (hdfsPath: String) 5. Packages all lower case, including package objects.

Abbreviation (don't do it!)	Dont abbreviate words in names, for example: • accountDate (not accountDt) • accountDateTime (not accountDtTm, and not accountDatetime) **Exceptions** • When using Id (for Identification or Identifier), do not shorten and use as lowerCamelCase, for example accountId
Split long lines of code	Long lines of code should be split over multiple lines. The Scala style guide suggests 80 characters as an upper bound, but discretion is allowed (R&D use 120 because they have big monitors!). If it doesn't fit on your screen without scrolling, then the line needs splitting!

General Scala/Spark	
Comments	If you need to comment your code to explain something, instead question whether you can write the code in a clearer way (generally the way to do this should result in more functional code). Good code should read like a story; if more detail is required, the developer can click through to see more information. A simple example code which follows this notion is as follows: Examples of where comments are normally required: 1. Where decisions are being made which can't be explained through code (e.g. features of the data and business feedback which determined how to write a score) 2. "TODO"s and "FIXME" comments (see below)

Scaladocs	See style guide. Scaladocs are effectively comments about what something does and for methods/functions the parameters it takes. Objects and key methods/functions should have Scaladocs explaining what they are/do. An example from the style guide:
	
	`/** Creates a person with a given name and birthdate`
	`*`
	`* @param name their name`
	`* @param birthDate the person's birthdate`
	`* @return a new Person instance with the age determined by the`
	`* birthdate and current date.`
	`*/`
	`def apply(name: String, birthDate: java.util.Date) = {}`
	` }`

General Scala/Spark	

TODO/FIXME	When something is left to do, use a TODO:
	
	//TODO: Add support for Longs
	
	When something may not work as expected, use a FIXME:
	
	//FIXME: Pattern match will fail if user provides Integer parameters
	
	In both cases, be **specific** about what the task is, as it may be a different developer who works on the task

Hard-coded configuration	Do not hard code configuration! Examples which should be externalised into a configuration project (e.g. type-safe) include: 1. Locations of raw data / output data 2. Iteration number 3. Dates 4. Scoring Parameters where the client wishes to be able to change these parameters
Regular Expressions	Regular expressions are compiled at run time and add significant overhead. As such, where possible they should be defined once only.

Spark only	
Broadcast joins	When one dataset in a join is small it is a good idea to suggest to Spark to broadcast the small dataset. largeDF.join(broadcast(smallDF), Seq("foo")) A broadcast join is sort of like a hash merge in SAS - instead of shuffling both datasets to do a join the smaller one is 'sent' to each executor which means the large dataset doesn't need to 'move'. This is more efficient.
Joins on same column(s)	**Use** df1.join(df2, Seq("id")) **Dont Use** df1.join(df2, df1("id") === df2("id")) This is only applicable where you are joining two tables on a column (or columns) with the same name. This avoids ending up with duplicate columns in the output dataframe and is much more succinct, especially when joining on multiple columns

Prefer select over drop	Instead of doing a long series of drops e.g. df.drop("a").drop("b").drop("c").drop("d").drop("e") consider selecting instead e.g. df.select("g","h","i","j","k") Selecting makes it clear what is on the output dataframe. Furthermore, Spark will not complain if you try and drop a column which doesn't exist (which in general is not desirable). Additional benefits of select over drop include being able to choose the order of the columns and also rename them in the same step: df.select($"g".as("moreLogicalName"),...)
Column referencing	There are 3 ways of referencing columns: • col("columnName") • $"columnName" • 'columnName $"columnName" is preferred as it is short but won't ruin code highlighting if you have code in a text editor such as notepad++. For this reason, avoid 'columnName Note that it is very common to do some coding in notepad++ on projects when doing ad-hoc coding work before moving it into an etl/scoring project.

Assessment 1

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 ($)

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
Dataset	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)

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


object AccountAssignment extends App {

  //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)

  //Create DataFrames of sources
  val customerDF: DataFrame = spark.read.option("header","true")
    .csv("src/main/resources/customer_data.csv")
  val accountDF = spark.read.option("header","true")
    .csv("src/main/resources/account_data.csv")

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

  //Expected Output Format
  case class CustomerAccountOutput(
                                   customerId: String,
                                   forename: String,
                                   surname: String,
                                   //Accounts for this customer
                                   accounts: Seq[AccountData],
                                   //Statistics of the accounts
                                   numberAccounts: Int,
                                   totalBalance: Long,
                                   averageBalance: Double
                                  )

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

  customerDS.show
  accountDS.show

//END GIVEN CODE
```

Assessment 2

A problem we have at Quantexa is where an address is populated with one string of text. In order to use this information in the Quantexa product, this field must be "parsed".

The purpose of parsing is to extract information from an entry - for example, to extract a forename and surname from a 'full_name' field. We will normally place the extracted fields in a case class which will sit as an entry in the wider case class; parsing will populate this entry.

You must join the given address data to the case class created in Assessment 1 and subsequently use the address parser provided to populate the parsed address fields in the Address case class.

Data

You have been provided with address data which contains the information described below

Address Data:

Field	Description
addressId	A unique ID for each individual address in the data.
customerId	A unique ID for each individual customer in the data.
address	The unparsed address.

The parquet file generated in Assessment 1 is required to proceed.

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	customerDocument	A Dataset, which contains elements of the CustomerDocument case class.

Table 1: Expected output format

Type	Name	Fields	Field Type	Description
Dataset	CustomerDocument	customerId forename surname accounts address	String String String Seq[AccountData] Seq[AddressData]	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 address data given into the hierarchical case class "CustomerDocument", and subsequently parse the address information using the parsing function provided. 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)

```
import com.quantexa.assignments.accounts.AccountAssignment.{AccountData,
CustomerAccountOutput}
import org.apache.log4j.{Level, Logger}
import org.apache.spark.sql.{DataFrame, SparkSession}


object CustomerAddress extends App {

  //Create a spark context, using a local master so Spark runs on the local
machine
  val spark =
SparkSession.builder().master("local[*]").appName("CustomerAddress").getOrCreate(
)

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

  import spark.implicits._

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

  case class AddressRawData(
                             addressId: String,
                             customerId: String,
                             address: String
                           )

  case class AddressData(
                          addressId: String,
                          customerId: String,
                          address: String,
                          number: Option[Int],
                          road: Option[String],
                          city: Option[String],
                          country: Option[String]
                        )

  //Expected Output Format
  case class CustomerDocument(
                               customerId: String,
                               forename: String,
                               surname: String,
                               //Accounts for this customer
                               accounts: Seq[AccountData],
                               //Addresses for this customer
                               address: Seq[AddressData]
                             )


  def addressParser(unparsedAddress: Seq[AddressData]): Seq[AddressData] = {
    unparsedAddress.map(address => {
      val split = address.address.split(", ")

      address.copy(
        number = Some(split(0).toInt),
        road = Some(split(1)),
        city = Some(split(2)),
        country = Some(split(3)))
    }
    )
  }
  val addressDF: DataFrame = spark.read.option("header",
"true").csv("src/main/resources/address_data.csv")

//  val customerAccountDS =
spark.read.parquet("src/main/resources/customerAccountOutputDS.parquet").as[Custo
merAccountOutput]

  //END GIVEN CODE

}
```

Assessment 3

Part of the Quantexa solution is to flag high risk countries as a link to these countries may be an indication of tax evasion.

For this question you are required to populate the flag in the destination case class "ScoringModel" where the customer has an address in the British Virgin Islands.

This flag must be then used to return the number of customers in the dataset that have a link to a British Virgin Islands address.

Data

The parquet file generated in Assessment 2 is the starting point of this assignment.

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	ScoringModel	A Dataset, which contains elements of the ScoringModel case class.

Table 1: Expected output format

Type	Name	Fields	Field Type	Description
Dataset	ScoringModel	customerId forename surname accounts address linkToBVI	String String String Seq[AccountData] Seq[AddressData] Boolean	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 populate the linkToBVI field with a true/false value and map the data given into the hierarchical case class "ScoringModel". 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)

```
import com.quantexa.assignments.customerAddresses.CustomerAddress.AddressData
import com.quantexa.assignments.accounts.AccountAssignment.AccountData
import org.apache.log4j.{Level, Logger}
import org.apache.spark.sql.SparkSession

object ScoringModel extends App {


  //Create a spark context, using a local master so Spark runs on the local
machine
  val spark =
SparkSession.builder().master("local[*]").appName("ScoringModel").getOrCreate()

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

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

  case class CustomerDocument(
                               customerId: String,
                               forename: String,
                               surname: String,
                               //Accounts for this customer
                               accounts: Seq[AccountData],
                               //Addresses for this customer
                               address: Seq[AddressData]
                             )

  case class ScoringModel(
                           customerId: String,
                           forename: String,
                           surname: String,
                           //Accounts for this customer
                           accounts: Seq[AccountData],
                           //Addresses for this customer
                           address: Seq[AddressData],
                           linkToBVI: Boolean
                         )


  //END GIVEN CODE
```