NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT 4305 - Big Data Architecture

Project: Part One - Spark

Asim Abazi, Mats E. Mollestad

02. March, 2021

Contents

0 Introduction

Before we head into solving the tasks we would like to present assumptions we have made.

Setup

We've chosen to solve the tasks using scala. Prerequisites to running our solution is to have:

- Java JDK Version ≥ 11 (might work with version $8 \geq$ but we haven't tested).
- Apache Spark Version $\geq 3.0.2$, accompanying Hadoop of version 2.7.
- Scala Version 2.12 or newer.
- SBT Build Tool Version 1.4.6 or newer.

Data Model

To better understand the tasks we chose to create an ER model of how we perceived the data. Some assumptions we've made:

- *Tags* is an array of strings to avoid creating another table.
- Fields with datetime use java.time.format.DateTimeFormatter pattern: "yyyy-MM-dd HH:mm:ss"
- PostTypeId is an ENUM where ‘QUESTIONS=1; ANSWERS=2;‘

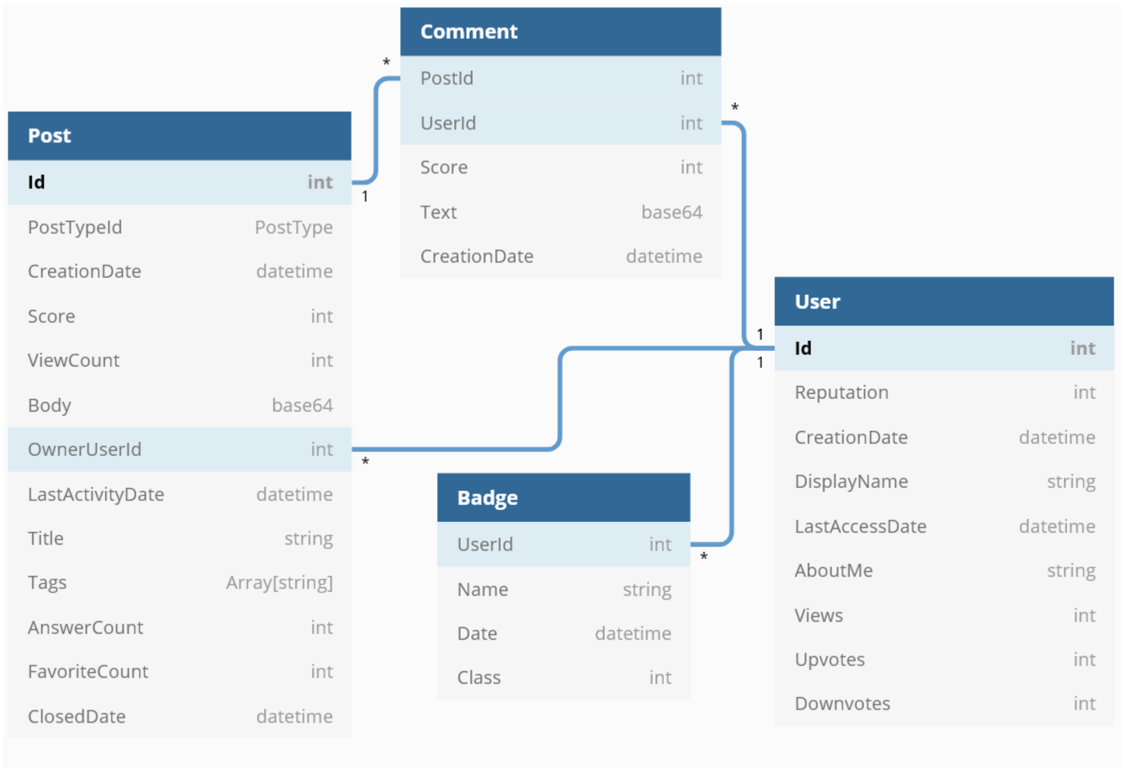

Figure 1: ER Diagram of Dataset

Processing Data

The datafiles accompanying this project were in comma-separated values (CSV) format. To solve the tasks we had to transform the data so we could easily process the data.

Loading the Data

We noticed the CSV values were delimited using whitespace, more spesifically tab. To load the values properly we:

- Loaded the file into a spark context
- Filtered row with column data e.g. ("UserId"\t"Name"\t"Date"\t"Class")
- Split rows (RDD[String]) into values (RDD[Array[String]]).

```
spark.sparkContext
      .textFile(config.badgesUri)
      .filter(x => !x.startsWith("\"UserId\"")).map(x => x.split("\t"))
```

Example of loading data/badges.csv

Process Objects

After loading the data we have it in the form of RDD[Array[String]], this means that to access a cell value you would have to index the array. Suffice it to say that this would be very tedious going forward. To deal with this we create processing objects that act as intermediaries between the raw data and the final processed output.

```
case class Badge(
    userId: Int,
    name: String,
    date: Option[LocalDateTime],
    classNumber: Int
)
object Badge {
    def fromRow(row: Array[String]): Badge = {
    Badge(
        row(0).toInt,
        row(1).toString,
        if (row(2) != "NULL") {
            Some(LocalDateTime.parse(row(2), MyUtils.dateFormatter))
            else None
        }
        row(3).toInt
    )
}
```

Example of Badges Processing Object

All that remains is to transform the inital RDD into a RDD with these processing objects. I.e. RDD[Array[String]] → RDD[Badges]. This is done with the fromRow method. Another benefit of doing this is that we can deal with null-values in the setup stage.

After we load all the RDDs we also make the processing objects.

We realize that using intermediary objects add some overhead, however the overhead was negligible compared to how much easier it was to work with objects.

```
val badges = badgesRdd.map(row => Badge.fromRow(row))
val posts = postsRdd.map(row => Post.fromRow(row))
val users = useresRdd.map(row => User.fromRow(row))
val comments = commentsRdd.map(row => Comment.fromRow(row))
```

Setting up Processing Objects

Running the Solution

We have tested two ways to run the solution.

- Running using SBT Build Tool from Command Line.
- Opening build.sbt from IDE (IntelliJ IDEA).

We recommend the first solution. Because it's much quicker and easier to setup argument passing

SBT Build Tool

(input_path).

1. Navigate to the project root. It should contain the build.sbt file.
2. Run "sbt compile".
3. Run "sbt 'run –input_path path/to/data'".

IntelliJ IDEA

1. Install Scala Plugin for IntelliJ
2. Press "Open" or "Open Project"
3. Navigate to the project root folder and press on build.sbt.

To make it work with IntelliJ you have to setup argument passing yourself.

Reading our Code

We've tried to keep the Main method of the program neat so that the read does not become overwhelmed by the amount of code. That being said we've generally made one method per task and put them all in the Task.scala file. The scala files are all under src/main/scala as per convention. We've chosen deliberately to keep comments at a minimum because we believe that functional code is very readable as long as the variable names are adequate. We have also done a fair bit of explanation of our code in this accompanying document.

1 Task

1.1 Load the posts.csv.gz into an RDD

We ended up loading the whole csv file as a raw text file, filtering out the header based on a prefix ”Id”, and then separating each line based on the tab symbol. This is shown in listing 4.

1.2 Load the comments.csv.gz into an RDD

The same process is done as in section 1.1, however here is the header filtered on ”PostId”.

1.3 Load the users.csv.gz into an RDD

The same process is done as in section 1.1, however here is the header filtered on ”Id”.

1.4 Load the badges.csv.gz into an RDD

The same process is done as in section 1.1, however here is the header filtered on ”UserId”.

```scala
object Setup {
  // Task 1.1
  def LoadBadgesRDD(spark: SparkSession, config: ProjectFileConfig): RDD[Array[String]] = {
    spark.sparkContext
      .textFile(config.badgesUri)
      .filter(x => !x.startsWith("\"UserId\"")).map(x => x.split("\t"))
  }

  def LoadPostsRDD(spark: SparkSession, config: ProjectFileConfig)
  : RDD[Array[String]] = {
    spark.sparkContext
      .textFile(config.postsUri)
      .filter(x => !x.startsWith("\"Id\"")).map(x => x.split("\t"))
  }

  def LoadCommentsRDD(spark: SparkSession, config: ProjectFileConfig): RDD[Array[String]] = {
    spark.sparkContext
      .textFile(config.commentsUri)
      .filter(x => !x.startsWith("\"PostId\"")).map(x => x.split("\t"))
  }

  def LoadUseresRDD(spark: SparkSession, config: ProjectFileConfig)
  : RDD[Array[String]] = {
    spark.sparkContext
      .textFile(config.usersUri)
      .filter(x => !x.startsWith("\"Id\"")).map(x => x.split("\t"))
  }
}
```

Loading posts.csv.gz

1.5 Count rows in each RDD

As each RDD has a .count() method, will this be used and printed to STDOUT. This is shown in listing 5.

```scala
object Task {
  def countRDDRows(postsRdd: RDD[Array[String]],
                   commentsRdd: RDD[Array[String]],
                   usersRdd: RDD[Array[String]],
                   badgesRdd: RDD[Array[String]]): Unit = {
    println("=== TASKS 1.1 through 1.4 START ===\n")
    println(s"Posts Rows: ${postsRdd.count()}")
    println(s"Comments Rows: ${commentsRdd.count()}")
    println(s"Users Rows: ${usersRdd.count()}")
    println(s"Badges Rows: ${badgesRdd.count()}")
    println("\n=== TASKS 1.1 through 1.4 END ===\n")
  }
}
```

Printing line count for each RDD

2 Task

2.1 Find the average length of the questions, answers, and comments in character.

We separated computation into it's own static method, as it was needed for separate RDD values shown in listing 6. This is then done on all the questions, answers and comments, which is shown in listing 7. Resulting in the output shown in section 2.1.

```
object Computations {

  def averageCharLength(strings: RDD[String]): Double = {
    val charLength = strings.map(string => {
      (string.toCharArray().length, 1)
    })
      .reduce((x, y) => (x._1 + y._1, x._2 + y._2))

    charLength._1.doubleValue() / charLength._2.doubleValue()
  }
}
```

Code for: Calculating the average char length for an RDD[String] type

```
  def averageCharacterLengthInTexts(posts: RDD[Post], comments: RDD[Comment]): Unit = {
    val questions = posts.filter(post => post.postTypeId == 1)
    val answers = posts.filter(post => post.postTypeId == 2)

    val averageAnswerCharLength = Computations.averageCharLength(answers.map(x => x.body))
    val averageCommentCharLength = Computations.averageCharLength(comments.map(x => x.text))
    val averageQuestionCharLength = Computations.averageCharLength(questions.map(x => x.title))

    println("=== Task 2.1 ===")
    println(s"Avg Answer length: $averageAnswerCharLength")
    println(s"Avg Question length: $averageQuestionCharLength")
    println(s"Avg Comment length: $averageCommentCharLength")
  }
```

Code for: Calculating the average char questions, answers, and comments

```
=== RESULT 2.1 ===
Avg Answer length: 1021.903266751286
Avg Question length: 57.22755812626051
Avg Comment length: 169.28073550693793
```

2.2 Find the dates when the first and the last questions were asked. Also, find the display name of users who posted those questions.

In order to get the oldest and newest posts would a reduce function fit well, combined with an if-statment. However, the logic is a bit more complicated, as we where dealing with an optional date value resulting in the code shown in listing 8. This will result in an algorithm working somewhat like a bubble-sort, where one comment is ”on the bubble” and always in danger of being swapped out.

```
val usersMap = users.map(user => (user.id, user.displayName))
  .collectAsMap()

val oldestPost = posts.filter(p => p.postTypeId == 1).reduce((oldestPost,
                                                          post) => {
  (post.creationDate, oldestPost.creationDate) match {
    case (Some(potential), Some(oldestDate)) =>
      if (potential.isBefore(oldestDate)) post else oldestPost
    case (Some(_), None) => post
    case _ => oldestPost
  }
})
val newestPost = posts.filter(p => p.postTypeId == 1).reduce((newestPost,
                                                          post) => {
  (post.creationDate, newestPost.creationDate) match {
    case (Some(potential), Some(newestDate)) =>
      if (newestDate.isBefore(potential)) post else newestPost
    case (Some(_), None) => post
    case _ => newestPost
  }
})
val newestPostUser = newestPost
  .ownerUserId
  .map(userID => usersMap(userID))
  .getOrElse("No User Found")

val oldestPostUser = oldestPost
  .ownerUserId
  .map(userID => usersMap(userID))
  .getOrElse("No User Found")
```

Code for: Ids of users with greatest number of answers and questions respectively code.

=== RESULT 2.2 ===
Newest Question Post Date: **2020-12-06T03:01:58**, by User: **mon**
Oldest Question Post Date: **2014-05-13T23:58:30**, by User: **Doorknob**

2.3 Find the ids of users who wrote the greatest number of answers and questions. Ignore the user with OwnerUserId equal to -1.

```
val mostQuestions = posts
  .filter(p => p.postTypeId == 1)
  .filter(p => p.ownerUserId.isDefined)
  .filter(p => p.ownerUserId.get != -1)
  .map(p => (p.ownerUserId.get, 1))
  .reduceByKey(_ + _)
  .sortBy(_._2, false)
  .take(1)
  .map { case (id, count) => println(s"(UserId:$id, count: $count)") }
val mostAnswers = posts
  .filter(p => p.postTypeId == 2)
  .filter(p => p.ownerUserId.isDefined)
  .filter(p => p.ownerUserId.get != -1)
  .map(p => (p.ownerUserId.get, 1))
  .reduceByKey(_ + _)
  .sortBy(_._2, false)
  .take(1)
  .map { case (id, count) => println(s"(UserId:$id, count: $count)") }
```

Code for: Ids of users with greatest number of answers and questions respectively code.

=== RESULT 2.3 ===
(UserId:**8820**, count: **103**)
===MOST QUESTION ONLY==
(UserId:**64377**, count: **579**)
===MOST ANSWERS ONLY==

2.4 Calculate the number of users who received less than three badges.

This was straight forward. Go through the badges and map userId and a count of 1, i.e. one occurence of the id. Then just reduce by key (userId) and keep values that are occur less than three times using the filter method.

The reduce is just addition.

```
val badgesLessThan3 = badges.map(badge => (badge.userId, 1))
    .reduceByKey(_ + _)
    .filter { case (_, badgeCount) => badgeCount < 3 }
    .count()
```

Code for: Badges less than 3.

=== RESULT 2.4: Users with less than three badges: **37189** ===

2.5 Calculate the Pearson correlation coefficient (or Pearson's r) between the number of upvotes and downvotes cast by a user.

1. Calculate upvotes mean.
2. Calculate downvotes mean.
3. Go through users and calculate $(x_i - \hat{x})(y_i - \hat{y})$, $(x_i - \hat{x})^2$ and $(y_i - \hat{y})$ respectively. This would be the equivalent of extending the users table with three extra columns.
4. Go through those three new columns and find the sum of them respecively. This would be the equivalent of calculating $\sum(x_i - \hat{x})(y_i - \hat{y})$, $\sum(x_i - \hat{x})^2$ and $\sum(y_i - \hat{y})^2$.
5. Finally square the denominator and compute *numerator/denominator*

```
val upvotesMean = users.map(u => u.upVotes).mean()
val downvotesMean = users.map(u => u.downVotes).mean()

val pearsonsTable = users.map(u => {
  val up = (u.upVotes - upvotesMean)
  val left = up * up
  val down = (u.downVotes - downvotesMean)
  val right = down * down
  val numerator = up * down

  (numerator, left, right)
}).collect()

val numeratorSum = pearsonsTable
  .map { case (numerator, _, _) => numerator }
  .sum
val left = pearsonsTable
  .map { case (_, left, _) => left }
  .sum
val right = pearsonsTable
  .map { case (_, _, right) => right }
  .sum
val denominator = left * right;

val res =
  (numeratorSum) / (math.sqrt(denominator))
```

Code for: Upvote and Downvote Pearson correlation coefficient.

=== Result 2.5: r-coefficient: **0.2684978771516632** ===

2.6 Calculate the entropy of id of users (that is UserId column from comments data) who wrote one or more comments.

1. Calculate the total number of comments.
2. Group all the comments based on the userId, in order to get the count.
3. Calculate the probability of a user making one of the comments.

$$P(x_i) = \frac{\text{number of comments by user}}{\text{total number of comments}}$$

   .

4. Then sum all the different probabilities using the following equation with a reduce.

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$$

   As Scala do not contain a native $\log_2$ formula, will this be calculated using $\log_{10}(x)/\log_{10}(2)$.

The code is shown in listing 12.

```scala
// Task 2.6
def userEntropy(comments: RDD[Comment]): Double = {
  val numberOfComments = comments.count()
  val userComments = comments.groupBy(comment => comment.userId).cache()

  val usersEntropy = userComments
    .map({ case (_, postIDs: Iterable[Comment]) =>
      postIDs.count(_ => true).toDouble / numberOfComments.toDouble
    })
    .reduce({ case (sum, value) => sum - value * log10(value) / log10(2) })

  println(s"=== Task 2.6: users comment entropy: ${usersEntropy} ===\n")

  usersEntropy
}
```

Code for: User comment entropy.

=== Task 2.6: users comment entropy: **11.257101558621223** ===

3 Task

3.1 Create a graph of posts and comments.

By mapping the data into a tuple (CommentUserID, PostUserID), and the weight between them being the amount of occurrences. This can cunstruct a Graph representing users commenting on other users posts.

In order to compute this, was a helper hash map constructed that makes it easy to get the creator of each post. Then all the users with a value of 0 or lower gets removed as they are not in the user csv file.

All the occurrences of the tuples are then counted and transformed into a RDD[Edge] type. At last will the different nodes be calculated by only mapping the user id to the VertexId. Shown with the code listing 13.

```scala
// Task 3.1
def userCommentGraph(
  comments: RDD[Comment],
  posts: RDD[Post],
  sparkContext: SparkContext
): Graph[Int, Long] = {
  val postOwner = posts.flatMap(post => post.ownerUserId.map(ownerId => (post.id, ownerId)))
    .collect()
    .toMap

  // Filtering on userId > 1 as some posts has
  val validUserComments = comments.filter(comment => comment.userId >= 1)
  val userComments = validUserComments.groupBy(comment => comment.userId).cache()

  val rawEdges = validUserComments.flatMap(comment => postOwner.get(comment.postId)
    .map(ownerId => (comment.userId, ownerId)))
    .countByValue()

  val edges = sparkContext.parallelize(
    rawEdges.map({ case (userIDs, count) => Edge(userIDs._1, userIDs._2, count) })
      .toSeq
  )

  val nodes: RDD[(VertexId, Int)] = userComments.map(user => (user._1, user._1))
  Graph(nodes, edges, -1)
}
```

Code for: Creating the graph representation.

3.2 Convert the result of the previous step into a Spark DataFrame.

In order to transform the graph to a DataFrame, was a tuple containing (CommentUserID, Amount Of Comments, PostOwnerUserID) created. At the same time was the dataframe labeled with a predefined constant, in order to remove the possibility of typos. The transformation is shown in code listing 14.

```
def dataframeFromGraph(graph: Graph[Int, Long], spark: SparkSession): DataFrame =
  spark.createDataFrame(graph.triplets.map(trip => (trip.srcId, trip.attr, trip.dstId)))
    .toDF(SQLStrings.commentUserID, SQLStrings.numberOfComments, SQLStrings.postUserID)
```

Code for: Transforming a Graph to DataFrame.

3.3 Find the user ids of top 10 users who wrote the most comments.

Following standard SQL-logic, resulted in grouping the dataframe on the CommentUserID, then summing all the NumberOfComments for all posts, sorting in descending order, and at last only returning the 10 first rows. Show in code listing 15.

```
// Task 3.3
def userIDsWithMostComments(dataframe: DataFrame, amount: Int = 10): Dataset[Row] = {

  println("=== Task 3.3 ===")

  val mostComments = dataframe.groupBy(col(SQLStrings.commentUserID))
    .agg(sum(SQLStrings.numberOfComments).as(SQLStrings.numberOfCommentsSum))
    .sort(col(SQLStrings.numberOfCommentsSum).desc)
    .limit(amount)

  mostComments.show()
  mostComments
}
```

Code for: UserIDs with the most comments.

CommentUserID	NumberOfCommentsSum
836	1263
381	1226
28175	864
64377	746
35644	693
55122	631
924	505
71442	493
21	406
45264	402

3.4 Find the display names of top 10 users who their posts received the greatest number of comments.

First was a DataFrame containing all user information created. This was easy as all our data was stored in case classes. We then went with a similar approach as in section 3.3, but this time we grouped on the PostOwnerUserID, joind the user data based on the user id's and then sorted the data. The code is shown in listing 16.

```
// Task 3.4
def usersWithMostCommentsOnTheirPost(
  dataframe: DataFrame,
  users: RDD[User],
  spark: SparkSession,
  amount: Int = 10
): Dataset[Row] = {

  val usersDataFrame = spark.createDataFrame(users).as(SQLStrings.usersTable)

  val mostComments = dataframe.groupBy(col(SQLStrings.postUserID))
    .agg(sum(SQLStrings.numberOfComments).as(SQLStrings.numberOfCommentsSum))
    .join(
      usersDataFrame,
      dataframe(SQLStrings.postUserID) === usersDataFrame(SQLStrings.userID)
    )
    .sort(col(SQLStrings.numberOfCommentsSum).desc)
    .limit(amount)

  println("=== Task 3.4 ===")
  mostComments.show()
  mostComments
}
```

Code for: Users with the most amount of comments on their posts.

PostUserID	NumberOfCommentsSum	reputation	displayName
836	875	24229	Neil Slater
64377	729	10346	Erwan
28175	695	11793	Media
45264	620	11711	n1k31t4
924	502	7248	Has QUIT–Anony-M...
29587	355	7613	JahKnows
65131	347	4611	Leevo
50727	318	4679	David Masip
80885	315	4782	Noah Weber
1330	312	10037	Brian Spiering

Output is partially truncated because it contains to much data to fit. Please refer to the solutions output if it's necessary to inspect the output properly.

3.5 Save the DF containing the information for the graph of posts and comments (from subtask 2) into a persistence format (like CSV) on your filesystem so that later could be loaded back into a Spark application's workspace.

We used the write method that exists for each DataFrame, and set the mode to overwrite, so it wont throw an error if the cvs file exists. We then use the url passed in when running and saving it to file_path/commentPosts.csv as shown in listing 17.

```
dataframe.write.mode("overwrite").csv(config.saveCommentPostUri)
```

Code for: Saving a DataFrame to a csv.