Supercomputing for Big Data – Lab Manual

R.P. Hes T.C. Leliveld

Contents

Introduction

In this lab we will put the concepts that are central to Supercomputing with Big Data in some practical context. We will analyze a large open data set and identify a way of processing it efficiently using Apache Spark and the Amazon Web Services (AWS). The data set in question is the GDelt 2.0 Global Knowledge Graph (GKG), which indexes persons, organizations, companies, locations, themes, and even emotions from live news reports in print, broadcast and internet sources all over the world. We will use this data to construct a histogram of the topics that are most popular on a given day, hopefully giving us some interesting insights into the most important themes in recent history.

Feedback is appreciated! The lab files will be hosted on GitHub. Feel free to make issues and/or pull requests to suggest or implement improvements.

Before You Start

The complete data set we will be looking at in lab 2 weighs in at several terabytes, so we need some kind of compute and storage infrastructure to run the pipeline. In this lab we will use Amazon AWS to facilitate this. As a student you are eligible for credits on this platform. We would like you to register for the GitHub Student Developer Pack, as soon as you decide to take this course. This gives you access to around 100 dollars worth of credits. This should be ample to complete lab 2. Note that you need a credit card to apply[1]. Don't forget to follow to register on AWS using the referral link from Github.

Make sure you register for these credits as soon as possible! You can always send an email to the TAs if you run into any trouble.

Before the end of the first week (Sunday 09/09/18), please send your TUDelft email address to the TAs to register for the lab

Goal of this Lab

The goal of this lab is to:

- familiarize yourself with Apache Spark, the MapReduce programming model, and Scala as a programming language;
- learn how to characterize your big data problem analytically and practically and what machines best fit this profile;
- get hands-on experience with cloud-based systems;
- learn about the existing infrastructure for big data and the difficulties with these; and
- learn how an existing application should be modified to function in a streaming data context.

You will work in groups of two. In this lab manual we will introduce a big data pipeline for identifying important events from the GDelt Global Knowledge Graph (GKG).

In lab 1, you will start by writing a Spark application that processes the GDelt dataset. You will run this application on a small subset of data on your local computer. You will use this to

1. get familiar with the Spark APIs,
2. analyze the application's scaling behavior, and
3. draw some conclusions on how to run it efficiently in the cloud.

It is up to you how you want to define *efficiently*, which can be in terms of performance, cost, or a combination of the two.

You may have noticed that the first lab does not contain any supercomputing, let alone big data. For lab 2, you will deploy your code on AWS, in an actual big data

[1] In case you don't have a credit card: In previous years, students have used prepaid credit cards (available online) to register.

cluster, in an effort to scale up your application to process the complete dataset, which measures several terabytes. It is up to you to find the configuration that will get you the most efficiency, as per your definition in lab 1.

For the final lab, we will modify the code from lab 1 to work in a streaming data context. You will attempt to rewrite the application to process events in real-time, in a way that is still scalable over many machines.

Lab 1

In this lab, we will design and develop the code in Spark to process GDelt data, which will be used in lab 2 to scale the analysis to the entire dataset. We will first give a brief introduction to the various technologies used in this lab.

Scala

Apache Spark, our big data framework of choice for this lab, is implemented in Scala, a compiled language on the JVM that supports a mix between functional and object-oriented programming. It is compatible with Java libraries. Some reasons why Spark was written in Scala are:

1. Compiling to the JVM makes the codebase extremely portable and deploying applications as easy as sending the Java bytecode (typically packaged in a **J**ava **AR**chive format, or JAR). This simplifies deploying to cloud provider big data platforms as we don't need specific knowledge of the operating system, or even the underlying architecture.

2. Compared to Java, Scala has some advantages in supporting more complex types, type inference, and anonymous functions[2]. Matei Zaharia, Apache Spark's original author, has said the following about why Spark was implemented in Scala in a Reddit AMA:

   > At the time we started, I really wanted a PL that supports a language-integrated interface (where people write functions inline, etc), because I thought that was the way people would want to program these applications after seeing research systems that had it (specifically Microsoft's DryadLINQ). However, I also wanted to be on the JVM in order to easily interact with the Hadoop filesystem and data formats for that. Scala was the only somewhat popular JVM language then that offered this kind of functional syntax and was also statically typed (letting us have some control over performance), so we chose that. Today there might be an argument to make the first version of the API in Java with Java 8, but we also benefitted from other aspects of Scala in Spark, like type inference, pattern matching, actor libraries, etc.

[2]Since Java 8, Java also supports anonymous functions, or lambda expression, but this version wasn't released at the time of Spark's initial release.

Apache Spark provides interfaces to Scala, R, Java and Python, but we will be using Scala to program in this lab. An introduction to Scala can be found on the Scala language site. You can have a brief look at it, but you can also pick up topics as you go through the lab.

Apache Spark

Apache Spark provides a programming model for a resilient distributed shared memory model. To elaborate on this, Spark allows you to program against a *unified view* of memory (i.e. RDD or DataFrame), while the processing happens *distributed* over *multiple nodes/machines/computers/servers* being able to compensate for *failures of these nodes*.

This allows us to define a computation and scale this over multiple machines without having to think about communication, distribution of data, and potential failures of nodes. This advantage comes at a cost: All applications have to comply with Spark's (restricted) programming model.

The programming model Spark exposes is based around the MapReduce paradigm. This is an important consideration when you would consider using Spark, does my problem fit into this paradigm?

Modern Spark exposes two APIs around this programming model:

1. Resilient Distributed Datasets
2. Spark SQL Dataframe/Datasets

We will consider both here shortly.

Resilient Distributed Datasets

RDDs are the original data abstraction used in Spark. Conceptually one can think of these as a large, unordered list of Java/Scala/Python objects, let's call these objects elements. This list of elements is divided in partitions (which may still contain multiple elements), which can reside on different machines. One can operate on these elements with a number of operations, which can be subdivided in wide and narrow dependencies, see tbl. 1. An illustration of the RDD abstraction can be seen in fig. 1.

RDDs are immutable, which means that the elements cannot be altered, without creating a new RDD. Furthermore, the application of transformations (wide or narrow) is lazy evaluation, meaning that the actual computation will be delayed until results are requested (an action in Spark terminology). When applying transformations, these will form a directed acyclic graph (DAG), that instructs workers what operations to perform, on which elements to find a specific result. This can be seen in fig. 1 as the arrows between elements.

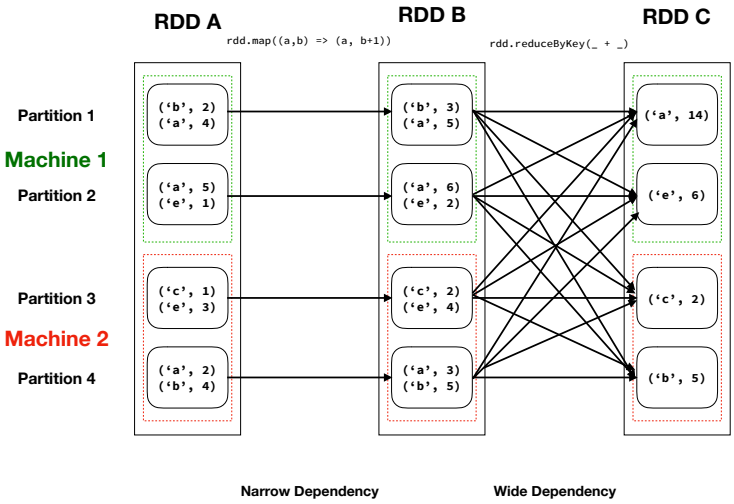

Figure 1: Illustration of RDD abstraction of an RDD with a tuple of characters and integers as elements.

Table 1: List of wide and narrow dependencies for (pair) RDD operations

Narrow Dependency	Wide Dependency
`map`	`coGroup`
`mapValues`	`flatMap`
`flatMap`	`groupByKey`
`filter`	`reduceByKey`
`mapPartitions`	`combineByKey`
`mapPartitionsWithIndex`	`distinct`
`join` with sorted keys	`join`
	`intersection`
	`repartition`
	`coalesce`
	`sort`

Now that you have an idea of what the abstraction is about, let's demonstrate some example code with the Spark shell. *If you want to paste pieces of code into the spark shell from this guide, it might be useful to copy from the github version, and use the* `:paste` *command in the spark shell to paste the code. Hit* `ctrl+D` *to stop pasting.*

```
$ spark-shell
2018-08-29 12:56:07 WARN  NativeCodeLoader:62 - Unable to load native-hadoop...
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR,...
Spark context Web UI available at http://
Spark context available as 'sc' (master = local[*], app id = local-1535540172727).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.1
```

Going back to our shell, let's first create some sample data that we can demonstrate the RDD API around. Here we create an infinite list of repeating characters from 'a' tot 'z'.

```
scala> val charsOnce = ('a' to 'z').toStream
charsOnce: scala.collection.immutable.Stream[Char] = Stream(a, ?)
scala> val chars: Stream[Char] = charsOnce #::: chars
chars: Stream[Char] = Stream(a, ?)
```

Now we build a collection with the first 200000 integers, zipped with the character stream. We display the first 30 results.

```
scala> val rdd = sc.parallelize(chars.zip(1 to 200000), numSlices=20)
rdd: org.apache.spark.rdd.RDD[(Char, Int)] =
                    ParallelCollectionRDD[0] at parallelize at <console>:26

scala> rdd.take(30)
res2: Array[(Char, Int)] = Array((a,1), (b,2), (c,3), (d,4), (e,5), (f,6),
(g,7), (h,8), (i,9), (j,10), (k,11), (l,12), (m,13), (n,14), (o,15), (p,16),
(q,17), (r,18), (s,19), (t,20), (u,21), (v,22), (w,23), (x,24), (y,25), (z,26),
(a,27), (b,28), (c,29), (d,30))
```

Let's dissect what just happened. We created a Scala object that is a list of tuples of `Chars` and `Ints` in the statement `(chars).zip(1 to 200000)`. With `sc.parallelize` we are transforming a Scala sequence into an RDD. This allows us to enter Spark's programming model. With the optional parameter `numSlices` we indicate in how many partitions we want to subdivide the sequence.

Let's apply some (lazily evaluated) transformations to this RDD.

```
scala> val mappedRDD = rdd.map({case (chr, num) => (chr, num+1)})
mappedRDD: org.apache.spark.rdd.RDD[(Char, Int)] =
                              MapPartitionsRDD[5] at map at <console>:25
```

We apply a `map` to the RDD, applying a function to all the elements in the RDD. The function we apply pattern matches over the elements as being a tuple of `(Char, Int)`, and add one to the integer. Scala's syntax can be a bit foreign, so if this is confusing, spend some time looking at tutorials and messing around in the Scala interpreter.

You might have noticed that the transformation completed awfully fast. This is Spark's lazy evaluation in action. No computation will be performed until an action is applied.

```
scala> val reducedRDD = rdd.reduceByKey(_ + _)
reducedRDD: org.apache.spark.rdd.RDD[(Char, Int)] =
                              ShuffledRDD[6] at reduceByKey at <console>:25
```

Now we apply a `reduceByKey` operation, grouping all of the identical keys together and merging the results with the specified function, in this case the `+` operator.

Now we will perform an action, which will trigger the computation of the transformations on the data. We will use the collect action, which means to gather all the results to the master, going out of the Spark programming model, back to a Scala sequence. How many elements do you expect there to be in this sequence after the previous transformations?

```
scala> reducedRDD.collect
res3: Array[(Char, Int)] = Array((d,769300000), (x,769253844), (e,769307693),
(y,769261536), (z,769269228), (f,769315386), (g,769323079), (h,769330772),
(i,769138464), (j,769146156), (k,769153848), (l,769161540), (m,769169232),
(n,769176924), (o,769184616), (p,769192308), (q,769200000), (r,769207692),
(s,769215384), (t,769223076), (a,769276921), (u,769230768), (b,769284614),
(v,769238460), (w,769246152), (c,769292307))
```

Typically, we don't build the data first, but we actually load it from a database or file system. Say we have some data in (multiple) files in a specific format. As an example consider `sensordata.csv` (in the `example` folder). We can load it as follows

```
// sc.textFile can take multiple files as argument!
scala> val raw_data = sc.textFile("sensordata.csv")
raw_data: org.apache.spark.rdd.RDD[String] =
                sensordata.csv MapPartitionsRDD[1] at textFile at <console>:24
scala> raw_data.take(10).foreach(println)
COHUTTA,3/10/14:1:01,10.27,1.73,881,1.56,85,1.94
COHUTTA,3/10/14:1:02,9.67,1.731,882,0.52,87,1.79
COHUTTA,3/10/14:1:03,10.47,1.732,882,1.7,92,0.66
COHUTTA,3/10/14:1:05,9.56,1.734,883,1.35,99,0.68
COHUTTA,3/10/14:1:06,9.74,1.736,884,1.27,92,0.73
COHUTTA,3/10/14:1:08,10.44,1.737,885,1.34,93,1.54
COHUTTA,3/10/14:1:09,9.83,1.738,885,0.06,76,1.44
COHUTTA,3/10/14:1:11,10.49,1.739,886,1.51,81,1.83
COHUTTA,3/10/14:1:12,9.79,1.739,886,1.74,82,1.91
COHUTTA,3/10/14:1:13,10.02,1.739,886,1.24,86,1.79
```

We can process this data to filter only measurements on `3/10/14:1:01`.

```
scala> val filterRDD = raw_data.map(_.split(","))
                    .filter(x => x(1) == "3/10/14:1:01")
filterRDD: org.apache.spark.rdd.RDD[Array[String]] =
                MapPartitionsRDD[11] at filter at <console>:25

scala> filterRDD.foreach(a => println(a.mkString(" ")))
COHUTTA 3/10/14:1:01 10.27 1.73 881 1.56 85 1.94
LAGNAPPE 3/10/14:1:01 9.59 1.602 777 0.09 88 1.78
```

```
NANTAHALLA 3/10/14:1:01 10.47 1.712 778 1.96 76 0.78
CHER 3/10/14:1:01 10.17 1.653 777 1.89 96 1.57
THERMALITO 3/10/14:1:01 10.24 1.75 777 1.25 80 0.89
ANDOUILLE 3/10/14:1:01 10.26 1.048 777 1.88 94 1.66
BUTTE 3/10/14:1:01 10.12 1.379 777 1.58 83 0.67
MOJO 3/10/14:1:01 10.47 1.828 967 0.36 77 1.75
CARGO 3/10/14:1:01 9.93 1.903 778 0.55 76 1.44
BBKING 3/10/14:1:01 10.03 0.839 967 1.17 80 1.28
```

You might have noticed that this is a bit tedious to work with, as we have to convert everything to Scala objects, and aggregations rely on having a pair RDD, which is fine when we have a single key, but for more complex aggregations, this becomes a bit tedious to juggle with.

Dataframe and Dataset

Our previous example is quite a typical use case for Spark. We have a big data store of some structured (tabular) format (be it csv, JSON, parquet, or something else) that we would like to analyse, typically in some SQL-like fashion. Manually applying operations to rows like this is both labour intensive, and inefficient, as we have knowledge of the 'schema' of data. This is where DataFrames originate from. Spark has an optimized SQL query engine that can optimize the compute path as well as provide a more efficient representation of the rows when given a schema. From the Spark SQL, DataFrames and Datasets Guide:

> Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API. When computing a result the same execution engine is used, independent of which API/language you are using to express the computation. This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation.

Under the hood, these are still immutable distributed collections of data (with the same compute graph semantics, only now Spark can apply extra optimizations because of the (structured) format.

Let's do the same analysis as last time using this API. First we will define a schema. Let's take a look at a single row of the csv:

```
COHUTTA,3/10/14:1:01,10.27,1.73,881,1.56,85,1.94
```

So first a string field, a date, a timestamp, and some numeric information. We can thus define the schema as such:

```
val schema =
  StructType(
    Array(
      StructField("sensorname", StringType, nullable=false),
      StructField("timestamp", TimestampType, nullable=false),
      StructField("numA", DoubleType, nullable=false),
      StructField("numB", DoubleType, nullable=false),
      StructField("numC", LongType, nullable=false),
      StructField("numD", DoubleType, nullable=false),
      StructField("numE", LongType, nullable=false),
      StructField("numF", DoubleType, nullable=false)
    )
  )
```

If we import types first, and then enter this in our interactive shell we get the following:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)
import org.apache.spark.sql.types._

val schema =
  StructType(
    Array(
      StructField("sensorname", StringType, nullable=false),
      StructField("timestamp", TimestampType, nullable=false),
      StructField("numA", DoubleType, nullable=false),
      StructField("numB", DoubleType, nullable=false),
      StructField("numC", LongType, nullable=false),
      StructField("numD", DoubleType, nullable=false),
      StructField("numE", LongType, nullable=false),
      StructField("numF", DoubleType, nullable=false)
    )
  )


// Exiting paste mode, now interpreting.

import org.apache.spark.sql.types._
schema: org.apache.spark.sql.types.StructType =
StructType(StructField(sensorname,StringType,false),
StructField(timestamp,TimestampType,false), StructField(numA,DoubleType,false),
StructField(numB,DoubleType,false), StructField(numC,LongType,false),
StructField(numD,DoubleType,false), StructField(numE,LongType,false),
StructField(numF,DoubleType,false))
```

An overview of the different Spark SQL types can be found online. For the timestamp field we need to specify the format according to the Java date format—in

our case `MM/dd/yy:hh:mm`. Tying this all together we can build a Dataframe like so.

```
scala> :paste
// Entering paste mode (ctrl-D to finish)
val df = spark.read
              .schema(schema)
              .option("timestampFormat", "MM/dd/yy:hh:mm")
              .csv("./sensordata.csv")
// Exiting paste mode, now interpreting.
df: org.apache.spark.sql.DataFrame =
        [sensorname: string, timestamp: date ... 6 more fields]

scala> df.printSchema
root
 |-- sensorname: string (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- numA: double (nullable = true)
 |-- numB: double (nullable = true)
 |-- numC: long (nullable = true)
 |-- numD: double (nullable = true)
 |-- numE: long (nullable = true)
 |-- numF: double (nullable = true

scala> df.take(10).foreach(println)
[COHUTTA,2014-03-10 01:01:00.0,10.27,1.73,881,1.56,85,1.94]
[COHUTTA,2014-03-10 01:02:00.0,9.67,1.731,882,0.52,87,1.79]
[COHUTTA,2014-03-10 01:03:00.0,10.47,1.732,882,1.7,92,0.66]
[COHUTTA,2014-03-10 01:05:00.0,9.56,1.734,883,1.35,99,0.68]
[COHUTTA,2014-03-10 01:06:00.0,9.74,1.736,884,1.27,92,0.73]
[COHUTTA,2014-03-10 01:08:00.0,10.44,1.737,885,1.34,93,1.54]
[COHUTTA,2014-03-10 01:09:00.0,9.83,1.738,885,0.06,76,1.44]
[COHUTTA,2014-03-10 01:11:00.0,10.49,1.739,886,1.51,81,1.83]
[COHUTTA,2014-03-10 01:12:00.0,9.79,1.739,886,1.74,82,1.91]
[COHUTTA,2014-03-10 01:13:00.0,10.02,1.739,886,1.24,86,1.79]
```

We can perform the same filtering operation as before in a couple of ways. We can use really error prone SQL queries (not recommended unless you absolutely love SQL and like debugging these command strings, this took me about 20 minutes to get right).

```
scala> df.createOrReplaceTempView("sensor")

scala> val dfFilter = spark.sql("SELECT * FROM sensor
WHERE timestamp=TIMESTAMP(\"2014-03-10 01:01:00\")")
// I think the newline in the multiline string breaks it if you paste it
dfFilter: org.apache.spark.sql.DataFrame =
            [sensorname: string, timestamp: timestamp ... 6 more fields]
```

```
scala> dfFilter.collect.foreach(println)
[COHUTTA,2014-03-10 01:01:00.0,10.27,1.73,881,1.56,85,1.94]
[NANTAHALLA,2014-03-10 01:01:00.0,10.47,1.712,778,1.96,76,0.78]
[THERMALITO,2014-03-10 01:01:00.0,10.24,1.75,777,1.25,80,0.89]
[BUTTE,2014-03-10 01:01:00.0,10.12,1.379,777,1.58,83,0.67]
[CARGO,2014-03-10 01:01:00.0,9.93,1.903,778,0.55,76,1.44]
[LAGNAPPE,2014-03-10 01:01:00.0,9.59,1.602,777,0.09,88,1.78]
[CHER,2014-03-10 01:01:00.0,10.17,1.653,777,1.89,96,1.57]
[ANDOUILLE,2014-03-10 01:01:00.0,10.26,1.048,777,1.88,94,1.66]
[MOJO,2014-03-10 01:01:00.0,10.47,1.828,967,0.36,77,1.75]
[BBKING,2014-03-10 01:01:00.0,10.03,0.839,967,1.17,80,1.28]
```

A slightly more sane and type-safe way would be to do the following.

```
scala> val dfFilter = df.filter("timestamp = TIMESTAMP(\"2014-03-10 01:01:00\")")
dfFilter: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
                    [sensorname: string, timestamp: timestamp ... 6 more fields]

scala> dfFilter.collect.foreach(println)
[COHUTTA,2014-03-10 01:01:00.0,10.27,1.73,881,1.56,85,1.94]
[NANTAHALLA,2014-03-10 01:01:00.0,10.47,1.712,778,1.96,76,0.78]
[THERMALITO,2014-03-10 01:01:00.0,10.24,1.75,777,1.25,80,0.89]
[BUTTE,2014-03-10 01:01:00.0,10.12,1.379,777,1.58,83,0.67]
[CARGO,2014-03-10 01:01:00.0,9.93,1.903,778,0.55,76,1.44]
[LAGNAPPE,2014-03-10 01:01:00.0,9.59,1.602,777,0.09,88,1.78]
[CHER,2014-03-10 01:01:00.0,10.17,1.653,777,1.89,96,1.57]
[ANDOUILLE,2014-03-10 01:01:00.0,10.26,1.048,777,1.88,94,1.66]
[MOJO,2014-03-10 01:01:00.0,10.47,1.828,967,0.36,77,1.75]
[BBKING,2014-03-10 01:01:00.0,10.03,0.839,967,1.17,80,1.28]
```

But this is still quite error-prone as writing these strings contains no typechecking. This is not a big deal when writing these queries in an interactive environment on a small dataset, but can be quite time consuming when there's a typo at the end of a long running job that means two hours of your (and the cluster's) time is wasted.

This is why the Spark community developed the Dataset abstraction. It is a sort of middle ground between Dataframes and RDDs, where you get some of the type safety of RDDs by operating on a case class (also known as product type). This allows us to use the compile-time typechecking on the product types, whilst still allowing Spark to optimize the query and storage of the data by making use of schemas.

Let's dive in some code, first we need to define a product type for a row.

```
scala> import java.sql.Timestamp
import java.sql.Timestamp

scala> :paste
// Entering paste mode (ctrl-D to finish)
```

```
case class SensorData (
    sensorName: String,
    timestamp: Timestamp,
    numA: Double,
    numB: Double,
    numC: Long,
    numD: Double,
    numE: Long,
    numF: Double
)

// Exiting paste mode, now interpreting.

defined class SensorData
```

Now we can convert a Dataframe (which actually is just an untyped Dataset) to a typed Dataset using the as method.

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val ds = spark.read .schema(schema)
              .option("timestampFormat", "MM/dd/yy:hh:mm")
              .csv("./sensordata.csv")
              .as[SensorData]

// Exiting paste mode, now interpreting.

ds: org.apache.spark.sql.Dataset[SensorData] =
            [sensorname: string, timestamp: timestamp ... 6 more fields]
```

Now we can apply compile time type-checked operations.

```
scala> val dsFilter = ds.filter(a => a.timestamp ==
                                  new Timestamp(2014 - 1900, 2, 10, 1, 1, 0, 0))
dsFilter: org.apache.spark.sql.Dataset[SensorData] =
                [sensorname: string, timestamp: timestamp ... 6 more fields]

scala> dsFilter.collect.foreach(println)
SensorData(COHUTTA,2014-03-10 01:01:00.0,10.27,1.73,881,1.56,85,1.94)
SensorData(NANTAHALLA,2014-03-10 01:01:00.0,10.47,1.712,778,1.96,76,0.78)
SensorData(THERMALITO,2014-03-10 01:01:00.0,10.24,1.75,777,1.25,80,0.89)
SensorData(BUTTE,2014-03-10 01:01:00.0,10.12,1.379,777,1.58,83,0.67)
SensorData(CARGO,2014-03-10 01:01:00.0,9.93,1.903,778,0.55,76,1.44)
SensorData(LAGNAPPE,2014-03-10 01:01:00.0,9.59,1.602,777,0.09,88,1.78)
SensorData(CHER,2014-03-10 01:01:00.0,10.17,1.653,777,1.89,96,1.57)
SensorData(ANDOUILLE,2014-03-10 01:01:00.0,10.26,1.048,777,1.88,94,1.66)
```

```
SensorData(MOJO,2014-03-10 01:01:00.0,10.47,1.828,967,0.36,77,1.75)
SensorData(BBKING,2014-03-10 01:01:00.0,10.03,0.839,967,1.17,80,1.28)
```

This provides us with more guarantees that are queries are valid (atleast on a type level).

This was a brief overview of the 2 (or 3) different Spark APIs. You can always find more information on the programming guides for RDDs and Dataframes/Datasets and in the Spark documentation

SBT

We showed how to run Spark in interactive mode. Now we will explain how to build applications, that can be submitted using the `spark-submit` command.

First, we will explain how to structure a Scala project, using SBT. The typical project structure is

```
├── build.sbt
└── src
    └── main
        └── scala
            └── example.scala
```

This is typical for JVM languages. Typically more directories are added under the `scala` folder to resemble the package structure.

The project's name, dependencies, and versioning is defined in the `build.sbt` file. An example `build.sbt` file is

```
ThisBuild / scalaVersion := "2.11.12"

lazy val example = (project in file("."))
  .settings(
    name := "Example project",
  )
```

This specifies the Scala version of the project (2.11.12) and the name of the project.

Open `example.scala` and add the following

```
package example


object Example {
  def main(args: Array[String]) {
    println("Hello world!")
  }
}
```

Run `sbt` in the root folder (the one where `build.sbt` is located). This puts you in interactive mode of SBT. We can compile the sources by writing the `compile` command.

```
$ sbt
[info] Loading project definition from ...
[info] Loading settings for project hello from build.sbt ...
[info] Set current project to Example project ...
[info] sbt server started at ...
sbt:Example project> compile
[success] Total time: 0 s, completed Sep 3, 2018 1:56:10 PM
```

We can try to run the application by typing `run`.

```
sbt:Example project> run
[info] Running example.Example
Hello world!
[success] Total time: 1 s, completed Sep 3, 2018 2:00:08 PM
```

Now let's add a function to `example.scala`.

```
object Example {
  def addOne(tuple: (Char, Int)) : (Char, Int) = tuple match {
    case (chr, int) => (chr, int+1)
  }
  def main(args: Array[String]) {
    println("Hello world!")
    println(addOne('a', 1))
  }
}
```

In your SBT session we can prepend any command with a tilde (~) to make them run automatically on source changes.

```
sbt:Example project> ~run
[info] Compiling 1 Scala source to ...
[info] Done compiling.
[info] Packaging ...
[info] Done packaging.
[info] Running example.Example
Hello world!
(a,2)
[success] Total time: 1 s, completed Sep 3, 2018 2:02:48 PM
1. Waiting for source changes in project hello... (press enter to interrupt)
```

We can also open an interactive session using SBT.

```
sbt:Example project> console
[info] Starting scala interpreter...
Welcome to Scala 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_102).
Type in expressions for evaluation. Or try :help.

scala> example.Example.addOne('a', 1)
res1: (Char, Int) = (a,2)

scala> println("Interactive environment")
Interactive environment
```

To build Spark applications with SBT we need to include dependencies (Spark most notably) to build the project. Modify your `build.sbt` file like so

```
ThisBuild / scalaVersion := "2.11.12"

lazy val example = (project in file("."))
  .settings(
    name := "Example project",

    libraryDependencies += "org.apache.spark" %% "spark-core" % "2.3.1",
    libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.3.1"
  )
```

We can now use Spark in the script. Modify `example.scala`.

```
package example

import org.apache.spark.sql.types._
import org.apache.spark.sql._
import java.sql.Timestamp


object ExampleSpark {
  case class SensorData (
    sensorName: String,
    timestamp: Timestamp,
    numA: Double,
    numB: Double,
    numC: Long,
    numD: Double,
    numE: Long,
    numF: Double
  )
  def main(args: Array[String]) {
    val schema =
      StructType(
        Array(
          StructField("sensorname", StringType, nullable=false),
```

```
          StructField("timestamp", TimestampType, nullable=false),
          StructField("numA", DoubleType, nullable=false),
          StructField("numB", DoubleType, nullable=false),
          StructField("numC", LongType, nullable=false),
          StructField("numD", DoubleType, nullable=false),
          StructField("numE", LongType, nullable=false),
          StructField("numF", DoubleType, nullable=false)
        )
      )

    val spark = SparkSession
      .builder
      .appName("Example")
      .getOrCreate()
    val sc = spark.sparkContext // If you need SparkContext object

    import spark.implicits._

    val ds = spark.read
                  .schema(schema)
                  .option("timestampFormat", "MM/dd/yy:hh:mm")
                  .csv("./sensordata.csv")
                  .as[SensorData]

    val dsFilter = ds.filter(a => a.timestamp ==
        new Timestamp(2014 - 1900, 2, 10, 1, 1, 0, 0))

    dsFilter.collect.foreach(println)

    spark.stop
  }
}
```

You can run the JAR via `spark-submit` (which will run on local mode).

```
$ spark-submit target/scala-2.11/example-project_2.11-0.1.0-SNAPSHOT.jar
INFO:...
SensorData(COHUTTA,2014-03-10 01:01:00.0,10.27,1.73,881,1.56,85,1.94)
SensorData(NANTAHALLA,2014-03-10 01:01:00.0,10.47,1.712,778,1.96,76,0.78)
SensorData(THERMALITO,2014-03-10 01:01:00.0,10.24,1.75,777,1.25,80,0.89)
SensorData(BUTTE,2014-03-10 01:01:00.0,10.12,1.379,777,1.58,83,0.67)
SensorData(CARGO,2014-03-10 01:01:00.0,9.93,1.903,778,0.55,76,1.44)
SensorData(LAGNAPPE,2014-03-10 01:01:00.0,9.59,1.602,777,0.09,88,1.78)
SensorData(CHER,2014-03-10 01:01:00.0,10.17,1.653,777,1.89,96,1.57)
SensorData(ANDOUILLE,2014-03-10 01:01:00.0,10.26,1.048,777,1.88,94,1.66)
SensorData(MOJO,2014-03-10 01:01:00.0,10.47,1.828,967,0.36,77,1.75)
SensorData(BBKING,2014-03-10 01:01:00.0,10.03,0.839,967,1.17,80,1.28)
INFO:...
```

By default, Spark's logging is quite assertive. You can change the log levels to warn to reduce the output.

For development purposes you can also try running the application from SBT using the `run` command. This is a bit iffy, as Spark starts a number of threads and these don't exit gracefully when SBT closes its main thread. This can be solved by running the application in a forked process, which can be enabled by setting `fork in run := true` in `build.sbt`. You will also have to set to change the log levels programmatically, if desired.

```
import org.apache.log4j.{Level, Logger}
...


def main(args: Array[String]) {
    ...
    Logger.getLogger("org.apache.spark").setLevel(Level.WARN)
    ...
}
```

You can also use this logger to log your application which might be helpful for debugging on the AWS cluster later on.

The GDelt Project

Now that we have introduced the different technologies, we can start talking about the goal of the first lab. In this lab you will write the application in Spark that analyzes GDelt and constructs the 10 most talked about topics per day. For the first lab you will write the prototype that you check on your local machine.

We will use the GDelt version 2 GKG files. The format these files are in is tab separated values. The exact specification of each columns and details can be found in the GKG codebook. The schema of the files can be read in `headers.csv`. The columns that are most relevant are the "date" column and the "allNames" column.

The entire dataset is split in 120164 csv files. We will provide you with a script that will download a number of sample files, as well as generate a file that can serve as an index where to load these files from.

Deliverables

The deliverables for the first lab are:

1. An RDD-based implementation,
2. A Dataframe/Dataset-based implementation,
3. A report outlining your approach and answers to the questions listed below.

Your report and code will be discussed in a brief oral examination during the lab, the schedule of which will be posted on Brightspace.

Questions

General questions:

1. Can you think of a problem/computation that does not fit Spark's MapReduce-esque programming model efficiently.
2. In typical use, what kind of operation would be more expensive, a narrow dependency or a wide dependency? Why?
3. Why do you think the MapReduce paradigm is such a widely utilized abstraction for distributed shared memory processing and fault-tolerance?
4. What is the shuffle operation and why is it such an important topic in Spark optimization?
5. In what way can Dataframes and Datasets improve performance both in compute, but also in the distributing of data compared to RDDs? Will Dataframes and Datasets always perform better than RDDs?
6. Consider the following scenario. You are running a Spark program on a big data cluster with 10 worker nodes and a single master node. One of the worker nodes fails. In what way does Spark's programming model help you recover the lost work? (Think about the directed acyclic graph!)

Implementation analysis questions:

1. Do you expect and observe big performance differences between the RDD and Dataframe/Dataset implementation of the GDelt analysis?
2. How will your application scale when increasing the amount of analyzed files? What will the progression in execution time be for, 100, 1000, 10000 files?
3. If you extrapolate the scaling behavior on your machine to the entire dataset, how much time will it take to process the entire dataset? Is this extrapolation reasonable on a single machine?
4. Now suppose you had a cluster of identical machines that you performed the analysis on. How many machines do you think you would need to process the entire dataset in under an hour?
5. Suppose you would run this analysis for a company. What do you think would be a appropriate way to measure the performance? Would it be the time it takes to execute? The amount of money it takes to perform the analysis on the cluster? A combination of these two, or something else? Pick something you think would be an interesting metric, as this is the metric you will be optimizing in the 2nd lab!