Big Data Analytics with Spark

Peter Prescott (201442927)

Introduction

Recent years have witnessed a *data revolution* characterised by unprecedented volume, variety and velocity (Kitchin 2014). The challenge of handling such quantities of data has led to the development of a series of new technologies: the *MapReduce* paradigm for distributed data processing (Dean and Ghemawat 2008), the *Hadoop* Distributed File System for storing and streaming such data (Shvachko et al. 2010), the *Spark* Resilient Distributed Dataset for in-memory cluster computing (Zaharia et al. 2012), and the Spark SQL extension's optimized relational *Dataframe* API (Armbrust et al. 2015).

For this assignment (Amen 2020), I am required to describe the middleware configuration of a Spark standalone cluster, to perform some simple analysis on a Spark Dataframe created from a CSV containing coronavirus data, and to present the results in a report of two A4 pages of 12-point text.

Middleware Configuration

Configuring a Spark cluster requires setting up Spark master and worker nodes running the same versions of Spark and Hadoop, and a PySpark driver running the same version of Python as that which is used to call it. To simplify matters, and to make it easy to reproduce the configuration across different machines, I used Docker, which is rapidly becoming accepted as the standard solution for reproducible research and collaborative software development (Boettiger 2015).

Docker allows the different components of an application (in this case, our Spark cluster) to be run in isolated virtual *containers* launched from

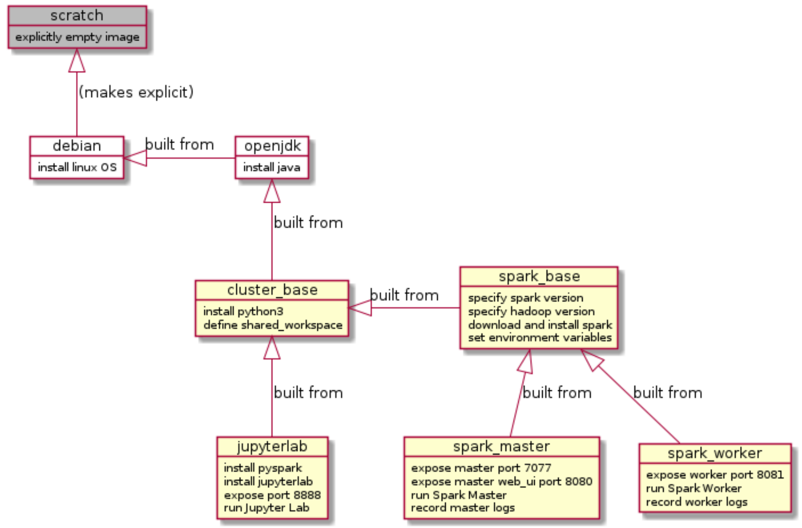

Figure 1: Docker Image Hierarchy

reproducible *images* defined explicitly by a 'Dockerfile' script. Based on the suggestion of Perez (2020), my configuration starts with a base image for the cluster, which adds a new installation of Python 3 to the pre-built publicly-available `openjdk:8-jre-slim` image, which offers a Java environment running on a Debian Linux kernel. On top of this, I defined a base Spark image for the master and worker images for the nodes of the Spark cluster; and separately an image for the PySpark driver, for which I set up the popular Jupyter notebook interface (Perez and Granger 2015). The relationship of the images to each other is shown in Figure 1. For full details, see the Dockerfiles on the project's git repository: github.com/pi-prescott/spark-standalone.

Once the images have been defined and built, the containers need to be run simultaneously with their network ports correctly mapped so that the different components of the configuration can interact successfully (Figure 2). These details are saved in a YAML configuration file, and then the cluster can be launched with a single command: `docker-compose up`.

Finally, we need to connect to the Spark cluster by initiating a SparkSession from our Jupyter notebook. We can then confirm everything is configured correctly by checking the Spark Master UI at localhost:8080.

Data Analytic Design

The assignment specifies a series of simple analytic tasks to perform: the data flow is visualized in Figure 3. First we read a CSV containing coronavirus data into a Spark dataframe; then we check it has the correct schema. We can ask Spark to automatically infer the schema, and it succeeds in distinguishing integers from strings, but not (in this case) in recognizing

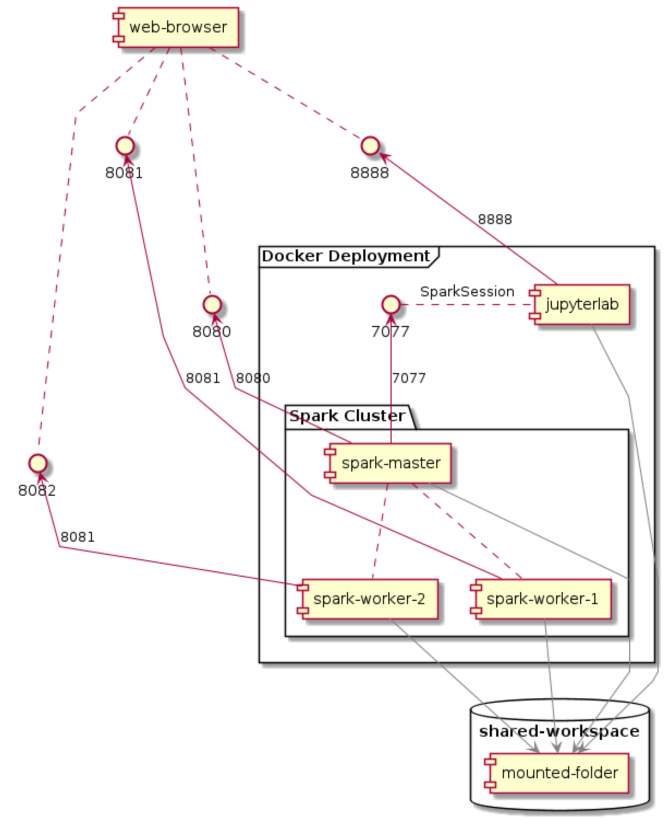

Figure 2: Standalone Configuration

that Date is a distinct type; so instead I specified the schema explicitly. We then use the filter function to remove null values, before using some other specific functions to find the highest `total_deaths` count in each country and the countries with highest and lowest `total_cases` counts.

Results and Discussion

It was found that the United States has the highest number of total_cases with 8,779,653, while Montserrat has the lowest, with 13. Confusingly, the assignment suggests that the highest total_deaths for Sweden should be 986, but this is the answer one will obtain if the schema is not correctly specified – the actual answer is 5,918. For the requested lists of twenty countries see the Jupyter notebook code and output in the Appendix.

Conclusions & Recommendations

This analysis was performed on a small CSV file of only 2.2 megabytes, which certainly does not qualify as *big data*. If we were to analyze a dataset of several terabytes then the single-machine standalone cluster we have configured would not be adequate. Instead we would need to actually leverage the capacity for utilizing a large cluster of distributed computing power that Spark is intended for – say by renting cloud computing power on AWS (AWS 2020). Because our configuration is precisely defined by Dockerfiles it should be straightforwardly transferable to such a scenario.

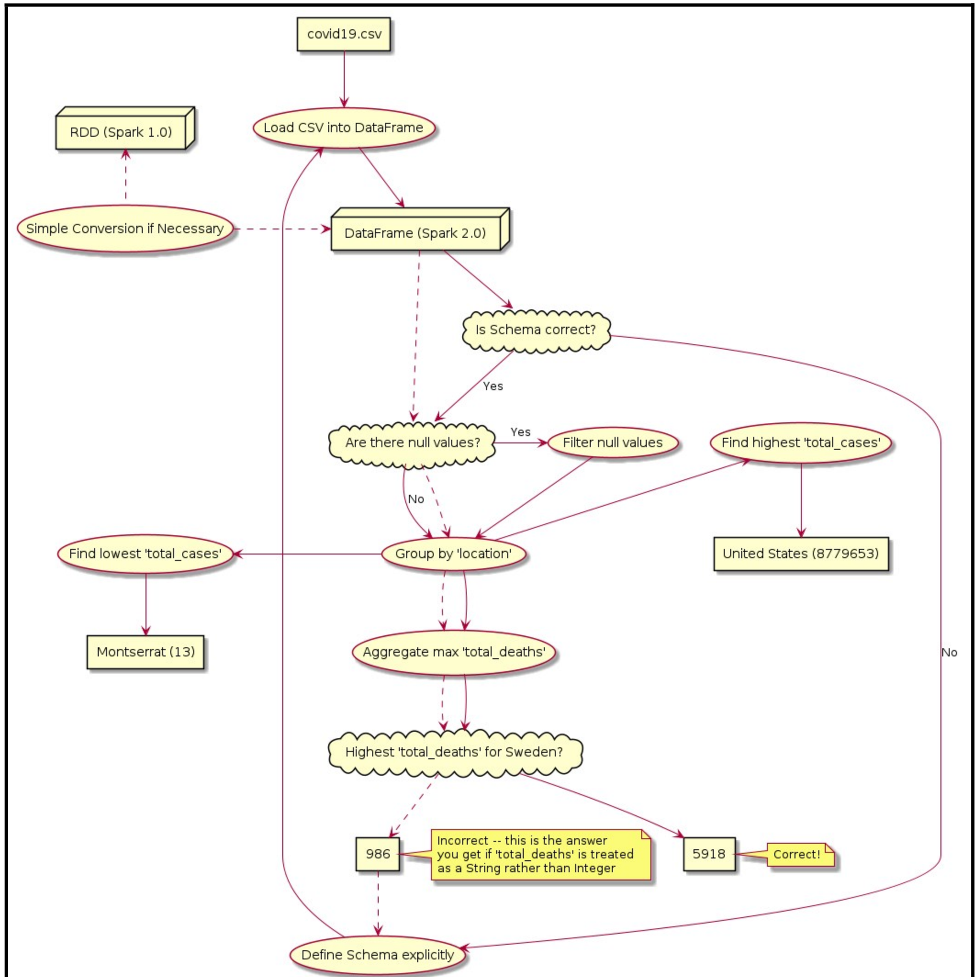

Figure 3: Data Flow Model

Our analysis has grouped the data by location to compare coronavirus numbers between countries, but without taking into account the differing sizes of these countries such comparisons are not very meaningful. It would also be interesting to consider the geospatial distribution of cases. Just as traditional database management systems have been extended with spatial database operations, similarly Sedona is a project currently incubating which extends Spark with *spatial* RDDs (Yu et al. 2019).

References

Amen, B. 2020. COMP529/336: Coursework Assignment #1 (Batch Analytics). Univeristy of Liverpool.

Armbrust, M.; Xin, R.S.; Lian, C.; Huai, Y.; Liu, D.; Bradley, J.K.; et al. 2015. Spark SQL: Relational Data Processing in Spark. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*: 1383–1394.

AWS. 2020. *Run Spark Applications with Docker Using Amazon EMR 6.x—Amazon EMR*. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-docker.html

Boettiger, C. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49: 71–79.

Dean, J.; Ghemawat, S. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51: 107–113.

Kitchin, R. 2014. *The Data Revolution: Big Data, Open Data, Data Infrastructures and Their Consequences*. SAGE Publications Ltd, Los Angeles, California.

Perez, A. 2020. Apache Spark Cluster on Docker (ft. A JuyterLab Interface). *Towards Data Science*.

Perez, F.; Granger, B.E. 2015. Project Jupyter: Computational narratives as the engine of collaborative data science..

Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. 2010. The Hadoop Distributed File System. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*: 1–10.

Yu, J.; Zhang, Z.; Sarwat, M. 2019. Spatial data management in Apache Spark: The GeoSpark perspective and beyond. *GeoInformatica* 23: 37–78.

Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauly, M.; et al. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*: 15–28.

Appendix: Python Code in Jupyter Notebook

(Available at github.com/pi-prescott/spark-standalone)

In [1]:
```
# [Q1] first we initiate a SparkSession with the spark-master

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("pyspark-notebook").\
        master("spark://spark-master:7077").\
        config("spark.executor.memory", "1024m").\
        getOrCreate()

spark
```

Out[1]: **SparkSession - in-memory**

SparkContext

Spark UI

Version	v3.0.0
Master	spark://spark-master:7077
AppName	pyspark-notebook

In [2]:
```
# [Q2a] load CSV into spark dataframe
# and [Q2b] check schema
wrong_schema = spark.read.csv(path='../data/covid19.csv',header=True)
# by default the type of each column is apparently assumed to be String.
print(wrong_schema.printSchema())
```

```
root
 |-- continent: string (nullable = true)
 |-- location: string (nullable = true)
 |-- date: string (nullable = true)
 |-- total_cases: string (nullable = true)
 |-- new_cases: string (nullable = true)
 |-- total_deaths: string (nullable = true)
 |-- new_deaths: string (nullable = true)

None
```

In [3]:
```
# if you ask explicitly, spark will try to infer the schema automatically
infer_schema = spark.read.csv(
    path='../data/covid19.csv',header=True,inferSchema=True)
infer_schema.printSchema()
# in this case it gets the integers right, but just treats the date as a string
```

```
root
 |-- continent: string (nullable = true)
 |-- location: string (nullable = true)
 |-- date: string (nullable = true)
 |-- total_cases: integer (nullable = true)
 |-- new_cases: integer (nullable = true)
 |-- total_deaths: integer (nullable = true)
 |-- new_deaths: integer (nullable = true)
```

In [4]:
```
# or you can specify the schema explicitly

from pyspark.sql.types import (StructField,
                               StringType,
                               IntegerType,
                               DateType,
                               StructType)

data_schema = [StructField('continent',StringType(),True),
              StructField('location',StringType(),True),
              StructField('date',DateType(),True),
              StructField('total_cases',IntegerType(),True),
              StructField('new_cases',IntegerType(),True),
              StructField('total_deaths',IntegerType(),True),
              StructField('new_deaths',IntegerType(),True)]

correct_struc = StructType(fields=data_schema)

dataframe = spark.read.csv(
    path='../data/covid19.csv', header=True, schema=correct_struc)

# and we can confirm that this time the types are correct
print(dataframe.printSchema())
```

```
root
 |-- continent: string (nullable = true)
 |-- location: string (nullable = true)
 |-- date: date (nullable = true)
 |-- total_cases: integer (nullable = true)
 |-- new_cases: integer (nullable = true)
 |-- total_deaths: integer (nullable = true)
 |-- new_deaths: integer (nullable = true)

None
```

In [5]:
```
# if we wanted to convert to the older-style RDD we easily could
rdd = dataframe.rdd
print(f'Created `rdd` {type(rdd)} from `dataframe` {type(dataframe)}.')
# ... and vice versa
new_dataframe = rdd.toDF()
print(f'Created `new_dataframe` {type(new_dataframe)}'
      + f' from `rdd` {type(rdd)}.')
```

```
Created `rdd` <class 'pyspark.rdd.RDD'> from `dataframe` <class 'pyspark.sql.data
frame.DataFrame'>.
Created `new_dataframe` <class 'pyspark.sql.dataframe.DataFrame'> from `rdd` <cla
ss 'pyspark.rdd.RDD'>.
```

In [6]:
```
# the simplest way to drop null values from a spark 2.0 dataframe
# ...is like this
drop_na = dataframe.dropna()
```

In [7]:
```
# [Q2c] but we can use the `.filter()` method if we like
filtered_df = dataframe.filter(
        ' and '.join([f'{x} is not null' for x in dataframe.columns])
        )
```

In [8]:
```
print(f'Before filtering we had {dataframe.count()} rows...')
print(f'Using `.dropna()` leaves us {drop_na.count()} rows.')
print(f'Using `.filter()` leaves us {filtered_df.count()} rows.')
if drop_na.count() == filtered_df.count():
    print('Good, those numbers are the same!')
else:
    print('Not good -- those numbers should be the same...')
```

```
Before filtering we had 53087 rows...
Using `.dropna()` leaves us 39974 rows.
Using `.filter()` leaves us 39974 rows.
Good, those numbers are the same!
```

In [9]:
```
# [Q3] use aggregate and groupBy functions to see highest `total_deaths` in each
hi_total_deaths = filtered_df.groupBy('location')\
                        .agg({'total_deaths':'max'})
```

In [10]:
```
hi_total_deaths.show()
```

```
+--------------------+----------------+
|            location|max(total_deaths)|
+--------------------+----------------+
|                Chad|              96|
|            Paraguay|            1327|
|              Russia|           26589|
|               Yemen|             600|
|             Senegal|             322|
|              Sweden|            5918|
|              Guyana|             119|
|              Jersey|              32|
|         Philippines|            7053|
|            Djibouti|              61|
|            Malaysia|             238|
|           Singapore|              28|
|                Fiji|               2|
|              Turkey|            9950|
|United States Vir...|              21|
|      Western Sahara|               1|
|              Malawi|             183|
|                Iraq|           10724|
|Sint Maarten (Dut...|              22|
|             Germany|           10183|
+--------------------+----------------+
only showing top 20 rows
```

In [11]:
```
# the assignment suggests that the number of total_deaths for Sweden should be 98
# however it is actually 5918
hi_total_deaths.filter(hi_total_deaths.location=='Sweden').show()
```

```
+--------+----------------+
|location|max(total_deaths)|
+--------+----------------+
|  Sweden|            5918|
+--------+----------------+
```

In [12]:
```
# however, we would get the result 986 if we hadn't
# explicitly made sure to load the CSV with the correct schema
wrong_schema.groupBy('location').agg({'total_deaths':'max'})\
            .filter(wrong_schema.location=='Sweden').show()
```

```
+--------+----------------+
|location|max(total_deaths)|
+--------+----------------+
|  Sweden|             986|
+--------+----------------+
```

In [13]:
```
# [Q4] use max and min functions to see which country
# has highest and lowest `total_cases`
# NB: 'total_cases' are given for every date,
# so for country with lowest can't simply find min(total_cases)
# as we'll get an earlier date with a lower figure
# rather than the country with the lowest final total_cases
# -- however, this is obviously not an issue for the maximum figure
import pyspark.sql.functions as F

filtered_df.select(F.max('total_cases')).show()
filtered_df.groupBy('location').max('total_cases')\
            .select(F.min('max(total_cases)')).show()
```

```
+----------------+
|max(total_cases)|
+----------------+
|         8779653|
+----------------+

+---------------------+
|min(max(total_cases))|
+---------------------+
|                   13|
+---------------------+
```

In [14]:
```
# to see a list of the countries with the highest and lowest total_cases count...
total_cases = filtered_df.groupBy('location').max('total_cases')
print('Countries with Highest Total Number of Cases')
total_cases.orderBy('max(total_cases)',ascending=False).show()
print('Countries with Lowest Total Number of Cases')
total_cases.orderBy('max(total_cases)',ascending=True).show()
```

```
Countries with Highest Total Number of Cases
+--------------+---------------+
|      location|max(total_cases)|
+--------------+---------------+
| United States|        8779653|
|         India|        7990322|
|        Brazil|        5439641|
|        Russia|        1547774|
|        France|        1198695|
|         Spain|        1116738|
|     Argentina|        1116596|
|      Colombia|        1033218|
|United Kingdom|         917575|
|        Mexico|         901268|
|          Peru|         892497|
|  South Africa|         717851|
|          Iran|         581824|
|         Italy|         564778|
|         Chile|         504525|
|       Germany|         464239|
|          Iraq|         459908|
|    Bangladesh|         401586|
|     Indonesia|         396454|
|   Philippines|         373144|
+--------------+---------------+
only showing top 20 rows

Countries with Lowest Total Number of Cases
+--------------------+---------------+
|            location|max(total_cases)|
+--------------------+---------------+
|          Montserrat|             13|
|                Fiji|             33|
|British Virgin Is...|             72|
|Northern Mariana ...|             92|
| Antigua and Barbuda|            124|
|              Brunei|            148|
|Bonaire Sint Eust...|            150|
|             Bermuda|            194|
|            Barbados|            233|
|      Cayman Islands|            239|
|            Guernsey|            266|
|              Monaco|            320|
|         Isle of Man|            352|
|           Mauritius|            439|
|       Liechtenstein|            483|
|            Tanzania|            509|
|             Comoros|            517|
|              Taiwan|            550|
|             Burundi|            558|
|              Jersey|            560|
+--------------------+---------------+
only showing top 20 rows
```