Demographic background of deceases in the United States

Group: "*HDFS*"

Group members: Alessio Ligios & Gianluca Visentin.

Indice

1. Introduction

The purpose of this document is to show the details of the second project of the "*Big Data*" course at RomaTre University.

The project aims to simulate a real system that takes input data and, after analyzing it, will save the results in the database.

The project is located in the GitHub usa-mortality-analysis repository and is structured as follows:

- <**data**>: this is the directory that contains database data; inside there are two directories representing the two nodes of the cluster, each of which contains the data saved in them
- <**README.md**>: it is the readme file of the project in which the contents of the repository are briefly described
- <**docker-compose.yml**>: it is the file that allows you to configure the database on the two nodes of the cluster through Docker
- <**usa-mortality-analysis.ipynb**>: it is the file containing the source code of the project, inside which the data are loaded, processed, the prediction analysis and the results saved in the database.

2. Dataset analysis

2.1. Dataset description

The dataset Death in the United States is a record of every death in the country for 2005 through 2015, including detailed information about causes of death and the demographic background of the deceased.

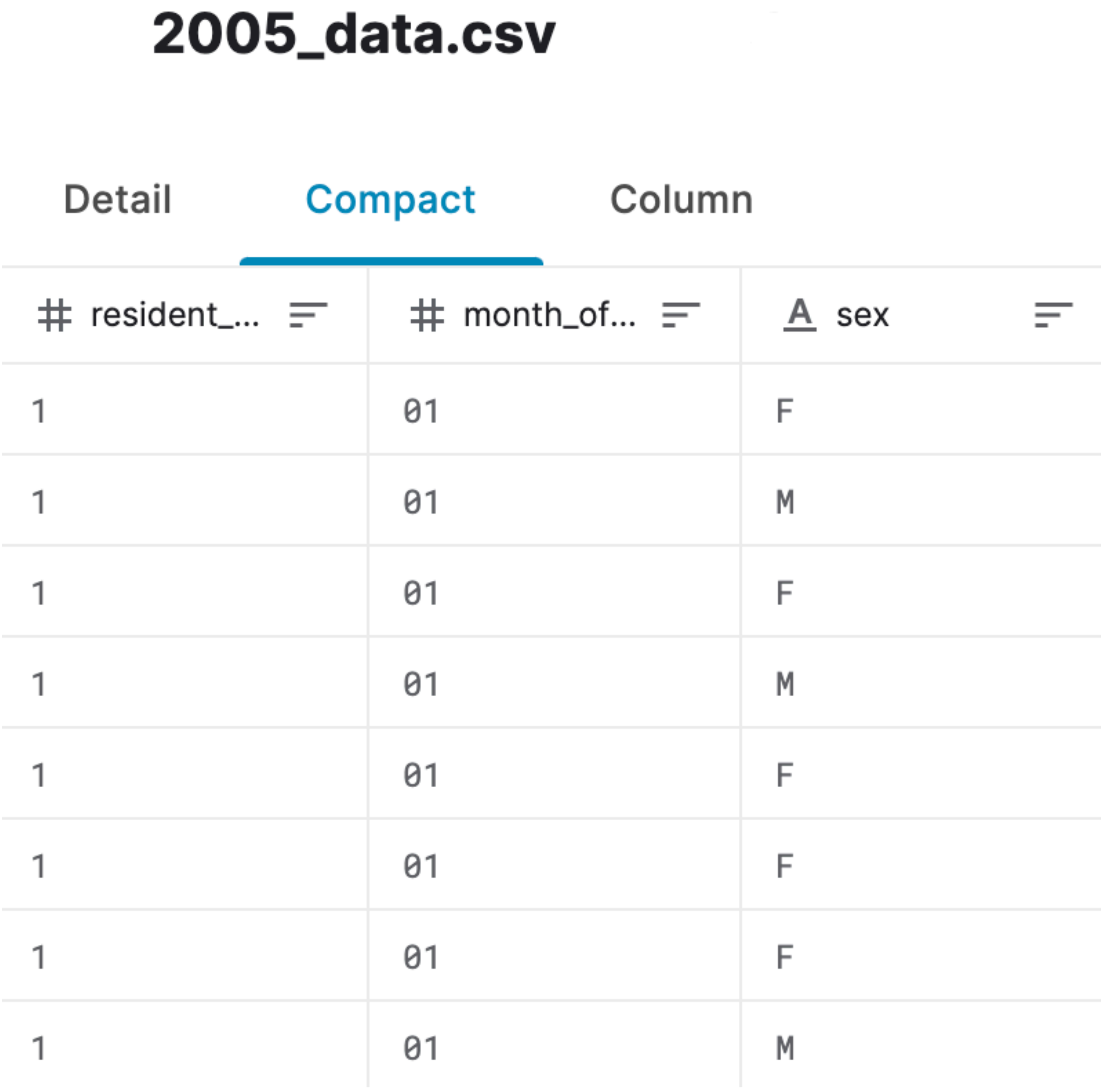
These data are extracted from the report into the deaths in the United States under the National Vital Statistics Systems, which is drawn up every year by the "Centers for Disease Control and Prevention" (CDC). Analyzing this topic is important for understanding the complex circumstances of death across the country.

The U.S. government uses this data to determine life expectancy and understand how death in the U.S. differs from the rest of the world.

2.2. Dataset structure

The dataset weighs about 4 GB and consists of 22 files, including 11 **csv** files that represent the data and 11 **json** files that represent the information encoded in the csv files. The *csv* files, with the corresponding *json* files, are classified according to the years from 2005 to 2015.

An example that better illustrates the organization of the dataset are the following portions of the <2005_data.csv> files and the corresponding <2005_codes.json>.


2005_data.csv

resident_...	month_of...	sex
1	01	F
1	01	M
1	01	F
1	01	M
1	01	F
1	01	F
1	01	F
1	01	M

2005_codes.json

About this file

2005

```
▼ "root" : { 34 items
    ▼ "resident_status" : { 4 items
        "1" : string "RESIDENTS"
        "2" : string "INTRASTATE NONRESIDENTS"
        "3" : string "INTERSTATE NONRESIDENTS"
        "4" : string "FOREIGN RESIDENTS"
    }
    ▶ "education_1989_revision" : {...} 12 items
    ▶ "education_2003_revision" : {...} 9 items
    ▶ "education_reporting_flag" : {...} 3 items
    ▼ "month_of_death" : { 12 items
        "10" : string "October"
        "11" : string "November"
        "12" : string "December"
        "01" : string "January"
        "02" : string "February"
        "03" : string "March"
```

The data in question refer to personal information concerning the deaths that occurred in that specific period of time. Types of information concern for example the state of residence (*resident-status*), the way the body was disposed after the death (*method-of-disposition*), the gender (*sex*), the month in which the death occurred (*month-of-death*), etc.

2.3. Data preprocessing

Usually, when we talk about data, we think of some large datasets with a huge number of rows and columns and above all with values that can be null or redundant.

Therefore, before proceeding to the data analysis phase, an initial **Data Preprocessing** phase is necessary, which is carried out by cleaning the "dirty" data and selecting the data that are of interest for the analysis in question.

Consequently, starting from a clean dataset improves the data extraction phase useful for subsequent transformations of the DataFrame.

The data processing was carried out through two operations:

- **Data filtering**: in the csv files there are 847 columns, of both *String* and *Integer* type. Within the scope of the project, only the columns of interest were selected, while all the others were eliminated from the dataset. This selection is made at the beginning of the job, when data from *csv* files are loaded and merged into a DataFrame.

```
# Dropping Columns
MergeData = MergeData.drop('record_condition_1',
                           'record_condition_2',
                           'record_condition_3',
                           'record_condition_4',
                           'record_condition_5',
                           'record_condition_6',
                           'record_condition_7',
                           'record_condition_8',
                           'record_condition_9',
                           'record_condition_10',
                           'record_condition_11',
                           'record_condition_12',
                           'record_condition_13',
                           'record_condition_14',
                           'record_condition_15',
                           'record_condition_16',
                           'record_condition_17',
                           'record_condition_18',
                           'record_condition_19',
                           'record_condition_20')
```

- **Data Cleaning**: job is also done, where the data are "cleaned" by filling the null values of columns according to policies based on heuristics.

```
# Null Imputations
MergeData = MergeData.fillna({'activity_code': 11})
MergeData = MergeData.fillna({'manner_of_death': 999})
MergeData = MergeData.fillna({'place_of_death_and_decedents_status': 999})
MergeData = MergeData.fillna({'education_2003_revision': 9})
```

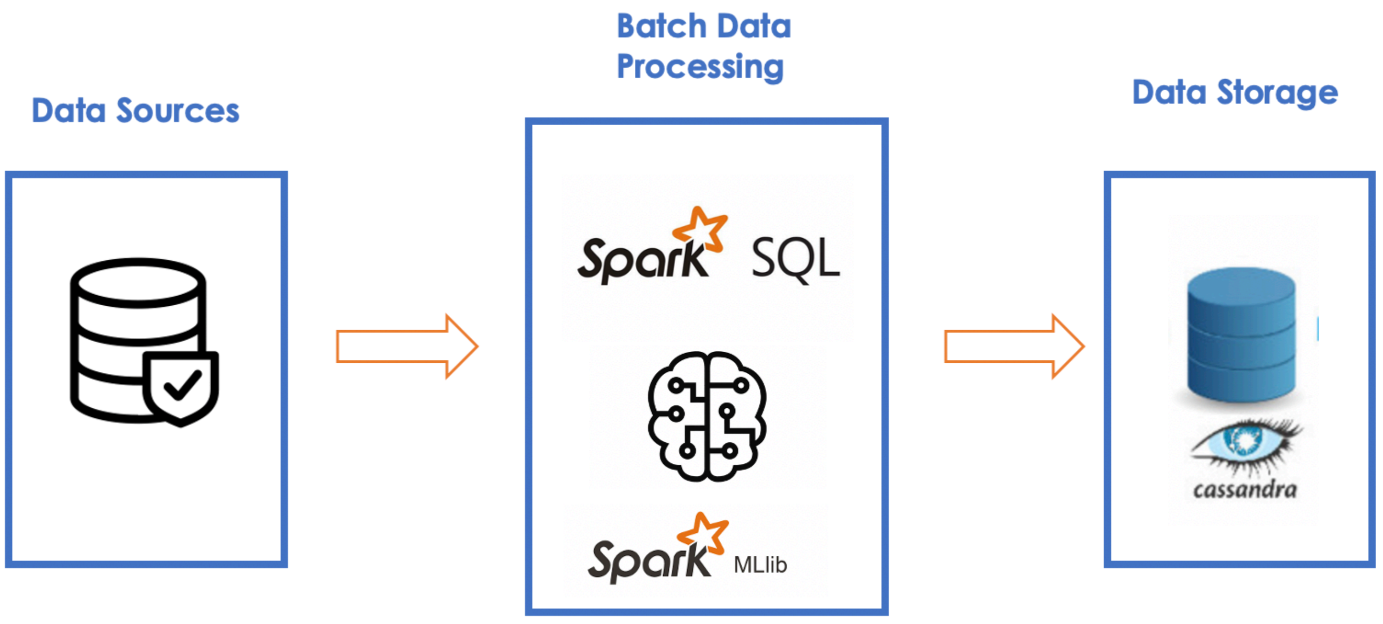
3. Goal of the project

The goal of the project is to implement a system capable of performing the pipeline of a traditional *Big Data* architecture, whose basic core is the **Batch Analysis** step.

In particular, we want to perform a series of analysis on the data available through technologies that allow the re-elaboration and saving of the analysis results in a database.

System features such as scalability, data availability, system partitioning in nodes of a cluster, related data replication and resulting fault tolerance are also considered.

4. Architecture



The architecture of the system is divided into 3 phases: in the first phase, called **Data Sources**, the files are acquired from the sources and prepared for analysis; in the second phase, called **Batch Data Processing**, the data prepared in the previous phase are taken and are analysed off-line through the use of various technologies such as *SparkSQL* or *Spark MLlib*; finally, in the third and final phase called **Data Storage** the results of the analysis carried out in the previous phase are saved and made persistent in a column-family database such as *Cassandra*.

The data in Cassandra is distributed between the two nodes of the cluster: in fact each node houses a *Docker* container environment that allows to run *Cassandra* with its default settings and without additional security settings.

5. Technologies

The technologies used in the system architecture are *SparkSQL* and *Spark MLlib* (based on *PySpark* language) for data analysis, *Cassandra* for managing the results provided by the results carried out from the analysis and *Docker* for creating a cluster environment.

5.1. PySpark

PySpark is a great language for performing exploratory data analysis at scale, building machine learning pipelines, and creating *ETLs* for a data platform.

Taking a step back, Spark is implemented in Scala, a language that runs on the JVM, so with PySpark it is possible to access all that functionality via Python.

You can think of *PySpark* as a Python-based wrapper on top of the Scala API, that communicates with the Spark Scala-based API via the Py4J library. Py4J isn't specific to PySpark or Spark. Py4J allows any Python program to talk to JVM-based code.

There are two reasons whys PySpark is based on the functional paradigm:

- Spark's native language, Scala, is functional-based.
- Functional code is much easier to parallelize.

Another way to think of *PySpark* is a library that allows processing large amounts of data on a single machine or a cluster of machines.

The fundamental concepts for the *PySpark* programming paradigm are:

- **SparkSession**: first of all, a Spark session needs to be initialized. With the help of SparkSession, DataFrame can be created and registered as tables. Moreover, SQL tables be executed, tables can be cached, and json/csv data formatted files can be read.
- **Spark Dataframes**: the key data type used in *PySpark*. This object can be thought of as a table distributed across a cluster and has functionality that is similar to dataframes in R and Pandas. If you want to do distributed computation using *PySpark*, then you'll need to perform operations on Spark dataframes, and not other python data types. It is also possible to use Pandas dataframes when using Spark, by calling toPandas() on a Spark dataframe, which returns a pandas object. One of the key differences between Pandas and Spark dataframes is eager versus lazy execution. In *PySpark*,

operations are delayed until a result is actually needed in the pipeline. This approach is used to avoid pulling the full data frame into memory and enables more effective processing across a cluster of machines. With Pandas dataframes, everything is pulled into memory, and every Pandas operation is immediately applied.

- **Transforming Data**: many different types of operations can be performed on Spark dataframes, much like the wide variety of operations that can be applied on Pandas dataframes. One of the ways of performing operations on Spark dataframes is via Spark SQL, which enables dataframes to be queried as if they were tables.

In the project the Spark configuration was performed as shown in the following code snippet, in which it was set the the master URL to connect to (local[2] to run locally with 4 cores) and the name for the application shown in the Spark web UI(usa-mortality-analysis):

```
# Spark configuration
sparkConf=SparkConf().setMaster("local[2]")\
                        .setAppName("usa-mortality-analysis")

spark=SparkSession.builder.config(conf=sparkConf).getOrCreate()
```

5.1.1. SparkSQL

SparkSQL is the Apache Spark module that integrates the possibility of expressing computations in relational language with Spark's functional APIs.

By generalizing the context, industries use Hadoop extensively to analyse their datasets. The reason is that the Hadoop framework is based on a simple programming model (MapReduce) and allows a scalable, flexible, fault tolerant and cost-effective processing solution. Here, the main concern is to maintain speed in processing large datasets in terms of waiting time between queries and waiting time to run the program.

Spark was introduced by the Apache Software Foundation to accelerate the Hadoop computational processing software process. It is good to remember that Spark is not a modified version of Hadoop and in reality it does not depend on Hadoop because it has its own cluster management. Hadoop is just one of the ways to implement Spark.

Consequently, in relation to the recent modules developed in the Hadoop ecosystem, *SparkSQL* offers a much closer link between the relational world and procedural computation, through the declarative APIs of the DataFrames that integrate perfectly with the Spark procedural code.

The main use that has been made of *SparkSQL* is to run SQL queries programmatically while running SQL functions and returns the result as a DataFrame.

A sample of such use of SparkSQL is as follows:

```
# Query 2 - Male vs female deaths by month of the year
results_male_deaths_month = spark.sql(
  """SELECT month_of_death,
            sex,
            count(sex) AS sex_counts
     FROM mergedTable
     GROUP BY month_of_death, sex
     ORDER BY month_of_death, sex""")
```

5.1.2. Spark MLlib

MLlib is a scalable *Machine Learning* library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, and underlying optimization primitives.

Spark MLLib seamlessly integrates with other Spark components such as Spark SQL, Spark Streaming, and DataFrames and is installed in the Databricks runtime. The library is usable in Java, Scala, and Python as part of Spark applications, so that you can include it in complete workflows.

In the project, the objective proposed concerns the prediction, through a linear regression phase, of the method of disposition, divided between burial and cremation. Algorithms work on the prediction of features; to obtain them you need to perform a process consisting of 3 key steps: extraction, transformation and selection.

Data extraction is performed through the SQL process, transformation through the use of three functions such as StringIndexer, OneHotEncoder and VectorAssembler. We use the VectorAssembler to combine all the feature columns into a single vector column.

In this case only the columns relating to the Method of Disposition are converted.

```
# Pipeline- String Indexing, encoding and Vector Assembling
stages = [] # stages in our Pipeline
for categoricalCol in categoricalColumns:
    # Category Indexing with StringIndexer
    stringIndexer = StringIndexer(inputCol=categoricalCol,
                                  outputCol=categoricalCol + "Index")

    # Use OneHotEncoder to convert categorical variables
    # into binary SparseVectors
    encoder = OneHotEncoder(inputCols=[stringIndexer.getOutputCol()],
                            outputCols=[categoricalCol + "classVec"])

    # Add stages.  These are not run here, but will run all at once later on.
    stages += [stringIndexer, encoder]
```

It was decided to implement a logistic regression technique, which is used when the dependent variable (target) is categorical. The two variables chosen and assigned along the axes concern the two method of disposition with the highest number of cases in the dataset: burial and cremation.

So, like the machine learning pipeline, after randomly split dataset into training and test set:

```
# Splitting the dataset into training set and test set
(trainingData, testData) = dataset.randomSplit([0.7, 0.3], seed=100)
```

we fit the Logistic Model, obtaining an evaluation about binary classification - burial or cremation - which allows us to reach a good performance about accuracy.

```
# Logistic Regression model for predicting method-of-disposition
# (burial vs cremation)

# Create initial LogisticRegression model
lr = LogisticRegression(labelCol="label", featuresCol="features", maxIter=10)

# Train model with Training Data
lrModel = lr.fit(trainingData)
```

5.3. Cassandra

Cassandra is a non-relational database management system distributed with an open source license and optimized for the management of large amounts of data.

Cassandra provides a key-value storage structure, in which the keys correspond to the values grouped in column family: a column family consists of a name that identifies it and an array of key-value pairs.

Each key-value pair identifies a row, the value in turn contains a list of values.

The families of columns made available by *Cassandra* are two:

- simple type
- super type

The super type can be represented as a family contained in another family, the top step of this hierarchy is the keyspace, which contains the specific column family set of an application.

```
# Connection with Cassandra db
cluster = Cluster(['127.0.0.1'], port= 9042)
session = cluster.connect('mykeyspace')
```

This database respects the **availability** and **partitioning** properties regarding *CAP theorem*. The nodes always make it possible to write and read their data.

The ownership of data availability ensures that the system chosen for data storage is **fault tolerance**, managed by automatically replicating the data on the two nodes of the cluster. This means that in case of failure of one node of the cluster there is a continuous availability from the other nodes. The high *fault tolerance* therefore increases the durability and scalability of this technology.

The partitioning property instead leads to choosing a system architecture called "*ring network*", in which each node is hierarchically equal to the others. In fact, there are no master-slave or similar configurations, thus allowing to have a no-single-point-offailure system, that is, capable of being available even after the fall of any node on the network. The **Peer-to-Peer** protocol allows communication through the various nodes of the ring.

Below is a code snippet that acts as an example on how to write data in Cassandra, in reference to the query shown above.

```
# Saving data in the db
session.execute("CREATE TABLE IF NOT EXISTS results_male_deaths_month
                                                    (ind int primary key,
                                                     month_of_death int,
                                                     sex varchar,
                                                     sex_counts int)")

stmt = session.prepare("INSERT INTO results_male_deaths_month
                                                    (ind,
                                                     month_of_death,
                                                     sex,
                                                     sex_counts)
                    VALUES (?, ?, ?, ?)")

for ind, item in results_male_deaths_month.toPandas().iterrows():
    results = session.execute(stmt, [ind,
                                     item['month_of_death'],
                                     item['sex'],
                                     item['sex_counts']])
```

5.4. Docker

The two-node cluster was set up through Docker, which provides an easy way to create an *Apache Cassandra* cluster.

Docker is a container technology that simplifies creation, deployment, shipping and running of applications. It enables to configure any application once and run it anywhere. Most of *Docker*'s benefits are a result of Dockers ability to isolate applications and their dependencies.

Containers provide a lightweight approach to virtualisation and the main goal of a container is to abstract away the application from the operating system.

Containers aka operating-system-level virtualization is a method of virtualization where the kernel of the operating system allows the existence of multiple user spaces. As a result, multiple user spaces share the same kernel. Virtualization at the operating system level provides a lightweight approach to application isolation.

In the project two sub-folders *<node1>* and *<node2>* of the *<data>* directory have been created, in each of which the data of node 1 and node 2 will persist, respectively.

In this architecture node 1 is the main node, while node 2 is the seed node.

The *<docker-compose.yml>* file was then created to configure the cluster.

```
services:
 cas1:
    container_name: cas1
    image: cassandra:latest
    volumes:
      - /Users/alessio/Documents/Projects/usa-mortality-analysis/data/node1:
        /var/lib/cassandra/data
    ports:
      - 9042:9042
    environment:
      - CASSANDRA_START_RPC=true
      - CASSANDRA_CLUSTER_NAME=MyCluster
      - CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch
      - CASSANDRA_DC=datacenter1
 cas2:
  container_name: cas2
  image: cassandra:latest
  volumes:
      - /Users/alessio/Documents/Projects/usa-mortality-analysis/data/node2:
        /var/lib/cassandra/data
  ports:
      - 9043:9042
  command: bash -c 'sleep 60;  /docker-entrypoint.sh cassandra -f'
  depends_on:
    - cas1
  environment:
      - CASSANDRA_START_RPC=true
      - CASSANDRA_CLUSTER_NAME=MyCluster
      - CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch
      - CASSANDRA_DC=datacenter1
      - CASSANDRA_SEEDS=cas1
```

6. Results

This section shows the **results** obtained from the various analysis processes carried out, in particular the queries implemented through *SparkSQL* with the related output data, and the evaluation of the model implemented in *SparkMLlib* are of interest.

It is good to note that in some cases, for computational reasons, the system does not show all the results obtained, but only the first rows.

6.1. Queries

1. Number of deaths by gender and resident status:

   ```
   results_male_female_resident_status = spark.sql(
   """SELECT resident_status,
           sex,
           count(sex) AS sex_counts
    FROM mergedTable
    GROUP BY resident_status, sex
    ORDER BY resident_status, sex""")
   ```

   With output:

   ```
   +---------------+---+----------+
   |resident_status|sex|sex_counts|
   +---------------+---+----------+
   |              1|  F|   7590600|
   |              1|  M|   7460761|
   |              2|  F|   1236340|
   |              2|  M|   1414999|
   |              3|  F|    210242|
   |              3|  M|    266829|
   |              4|  F|     10052|
   |              4|  M|     20991|
   +---------------+---+----------+
   ```

2. Number of deaths by gender and month of the year:

   ```
   results_male_deaths_month = spark.sql(
   """SELECT month_of_death,
           sex,
           count(sex) AS sex_counts
    FROM mergedTable
    GROUP BY month_of_death, sex
    ORDER BY month_of_death, sex""")
   ```

   With output:

```
+--------------+---+----------+
|month_of_death|sex|sex_counts|
+--------------+---+----------+
|             1|  F|    849629|
|             1|  M|    838492|
|             2|  F|    758754|
|             2|  M|    750646|
|             3|  F|    813433|
|             3|  M|    804654|
|             4|  F|    747380|
|             4|  M|    754604|
|             5|  F|    738769|
|             5|  M|    753111|
|             6|  F|    696060|
|             6|  M|    717974|
|             7|  F|    714427|
|             7|  M|    740472|
|             8|  F|    711261|
|             8|  M|    735864|
|             9|  F|    700249|
|             9|  M|    719378|
|            10|  F|    749339|
|            10|  M|    763009|
+--------------+---+----------+
only showing top 20 rows
```

3. Top 20 causes of death by sex:

```
results_diseases_causing_deaths_sex = spark.sql(
"""SELECT m.sex AS sex,
        m.icd_code_10th_revision AS code,
        i.description3 AS description,
        count(m.sex) AS sex_counts
 FROM mergedTable m, icd10 i
 WHERE i.code3 = m.icd_code_10th_revision
 GROUP BY i.description3 ,m.icd_code_10th_revision, m.sex
 ORDER BY count(m.sex) DESC, m.sex
 LIMIT 20""")
```

With output:

```
+---+----+--------------------+----------+
|sex|code|         description|sex_counts|
+---+----+--------------------+----------+
|  M|I219|Acute myocardial ...|    487319|
|  F|G309| Alzheimer's disease|    415393|
|  F|J449|Other chronic obs...|    397198|
|  F|I219|Acute myocardial ...|    392758|
|  M|J449|Other chronic obs...|    350659|
|  M| C61|Malignant neoplas...|    204566|
|  M|G309| Alzheimer's disease|    184358|
|  F|J189|Pneumonia, unspec...|    173403|
|  M|J189|Pneumonia, unspec...|    151422|
|  M|C189|Malignant neoplas...|    150469|
|  F|C189|Malignant neoplas...|    144516|
|  M|C259|Malignant neoplas...|    137235|
|  F|C259|Malignant neoplas...|    132971|
|  F|A419|Sepsis, unspecifi...|    121247|
|  M|A419|Sepsis, unspecifi...|    104677|
|  M|I119|Hypertensive hear...|     98173|
|  M| G20| Parkinson's disease|     97789|
|  M|C159|Malignant neoplas...|     81824|
|  F| I10|Essential (primar...|     80630|
|  F|I119|Hypertensive hear...|     79545|
+---+----+--------------------+----------+
```

4. Method of disposition per year:

```
results_disposition = spark.sql(
"""SELECT current_data_year AS year,
        CASE method_of_disposition
        WHEN 'C' THEN 'Cremation'
        WHEN 'B' THEN 'Burial'
        WHEN 'D'THEN 'Donation'
        WHEN 'E' THEN 'Entombment'
        WHEN 'O' THEN 'Other'
        WHEN 'R' THEN 'RemovedFromUSA'
        WHEN 'U' THEN 'Unknown'
        END AS disposition,
        COUNT(*) AS counts
 FROM mergedTable
 GROUP BY 1, 2
 ORDER BY 1, 3""")
```

With output:

```
+----+-----------+-------+
|year|disposition| counts|
+----+-----------+-------+
|2005|  Cremation| 350018|
|2005|     Burial| 553202|
|2006|  Cremation| 423282|
|2006|     Burial| 667169|
|2007|  Cremation| 472220|
|2007|     Burial| 725666|
|2008|  Cremation| 579827|
|2008|     Burial| 866384|
|2009|  Cremation| 599202|
|2009|     Burial| 802305|
|2010|  Cremation| 706224|
|2010|     Burial| 906430|
|2011|  Cremation| 780480|
|2011|     Burial| 950372|
|2012|  Cremation| 898222|
|2012|     Burial|1093628|
|2013|  Cremation| 973768|
|2013|     Burial|1113362|
|2014|  Cremation|1094292|
|2014|     Burial|1162836|
+----+-----------+-------+
only showing top 20 rows
```

5. Number of deaths by cause, grouped by month of year:

```
results_deaths_month = spark.sql(
"""SELECT month_of_death AS month,
        CASE manner_of_death
        WHEN '0' THEN 'Not Specified'
        WHEN '1' THEN 'Accident'
        WHEN '2' THEN 'Suicide'
        WHEN '3' THEN 'Homicide'
        WHEN '4' THEN 'Pending investigation'
        WHEN '5' THEN 'Could not be determine'
        WHEN '6' THEN 'Self-Inflicted'
        WHEN '7' THEN 'Natural'
        ELSE 'OTHER'
        END AS manner_death,
        COUNT(*) AS counts
 FROM mergedTable
 GROUP BY 1, 2
 ORDER BY 1,2""")
```

With output:

```
+-----+--------------------+-------+
|month|         manner_death| counts|
+-----+--------------------+-------+
|    1|            Accident|  73168|
|    1|Could not be dete...|   6569|
|    1|            Homicide|  10496|
|    1|             Natural|1265741|
|    1|               OTHER| 305799|
|    1|Pending investiga...|   3025|
|    1|             Suicide|  23323|
|    2|            Accident|  67509|
|    2|Could not be dete...|   6161|
|    2|            Homicide|   8666|
|    2|             Natural|1128534|
|    2|               OTHER| 275216|
|    2|Pending investiga...|   2404|
|    2|             Suicide|  20910|
|    3|            Accident|  72695|
|    3|Could not be dete...|   6817|
|    3|            Homicide|  10283|
|    3|             Natural|1211157|
|    3|               OTHER| 290285|
|    3|Pending investiga...|   2645|
+-----+--------------------+-------+
only showing top 20 rows
```

6. Deaths caused by Alzheimer's by age groups:

```
results_alzheimer = spark.sql(
"""SELECT count(*) as deaths_counts,
        CASE age_recode_12
        WHEN '10' THEN '75 - 84 years'
        WHEN '11' THEN '85 years and over'
        WHEN '12' THEN 'Age not stated'
        WHEN '01' THEN 'Under 1 year'
        WHEN '02' THEN '1 - 4 years'
        WHEN '03' THEN '5 - 14 years'
        WHEN '04' THEN '15 - 24 years'
        WHEN '05' THEN '25 - 34 years'
        WHEN '06' THEN '35 - 44 years'
        WHEN '07' THEN '45 - 54 years'
        WHEN '08' THEN '55 - 64 years'
        WHEN '09' THEN '65 - 74 years'
        END AS age
 FROM mergedTable
 WHERE 113_cause_recode = '052'
 GROUP BY age
 ORDER BY deaths_counts DESC
 LIMIT 10""")
```

With output:

```
+-------------+-----------------+
|deaths_counts|              age|
+-------------+-----------------+
|       400844|85 years and over|
|       174941|    75 - 84 years|
|        32518|    65 - 74 years|
|         5763|    55 - 64 years|
|          700|    45 - 54 years|
|           64|    35 - 44 years|
|            6|   Age not stated|
|            3|    25 - 34 years|
|            1|    15 - 24 years|
+-------------+-----------------+
```

7. Number of suicides by educational level:

```
results_degree_suicides = spark.sql(
"""SELECT count(*) as deaths_counts,
        CASE education_2003_revision
        WHEN '1' THEN '8th grade or less'
        WHEN '2' THEN '9 - 12th grade, no diploma'
        WHEN '3' THEN 'high school graduate or GED completed'
        WHEN '4' THEN 'some college credit, but no degree'
        WHEN '5' THEN 'Associate degree'
        WHEN '6' THEN 'Bachelor’s degree'
        WHEN '7' THEN 'Master’s degree'
        WHEN '8' THEN 'Doctorate or professional degree'
        WHEN '9' THEN 'Unknown'
        END AS education
 FROM mergedTable
 WHERE manner_of_death = '2'
 GROUP BY education
 ORDER BY deaths_counts DESC
 LIMIT 10""")
```

With output:

```
+-------------+--------------------+
|deaths_counts|           education|
+-------------+--------------------+
|       113618|high school gradu...|
|        49674|some college cred...|
|        35932|9 - 12th grade, n...|
|        33985|   Bachelor’s degree|
|        19862|    Associate degree|
|        11797|   8th grade or less|
|        11095|     Master’s degree|
|         6076|             Unknown|
|         5657|Doctorate or prof...|
+-------------+--------------------+
```

8. Total number of suicides per year:

```
results_suicides = spark.sql(
"""SELECT current_data_year AS year,
        CASE manner_of_death
        WHEN '2' THEN 'Suicide'
        END AS suicides,
        COUNT(*) AS counts
 FROM jointTable
 GROUP BY 1, 2
 ORDER BY 1, 3""")
```

With output:

```
+----+--------+------+
|year|suicides|counts|
+----+--------+------+
|2005| Suicide| 32934|
|2006| Suicide| 33562|
|2007| Suicide| 34827|
|2008| Suicide| 36251|
|2009| Suicide| 37205|
|2010| Suicide| 38710|
|2011| Suicide| 39878|
|2012| Suicide| 40929|
|2013| Suicide| 41509|
|2014| Suicide| 43139|
|2015| Suicide| 44417|
+----+--------+------+
```

9. Number of deaths due to death declared after an investigative process:

```
results_deaths_pending = spark.sql(
"""SELECT icd.description3 AS description,
          count(*) as counts
   FROM mergedTable mt JOIN icd10 icd ON icd.code3 = mt.icd_code_10th_revision
   WHERE mt.manner_of_death == '4'
   GROUP BY description
   ORDER BY counts DESC, description""")
```

With output:

```
+--------------------+------+
|         description|counts|
+--------------------+------+
|Ill-defined and u...| 29259|
|Acute myocardial ...|   212|
|Hypertensive hear...|   153|
|Complications and...|   148|
|      Cardiac arrest|   102|
|Other chronic obs...|    94|
|      Cardiomyopathy|    87|
|Pneumonia, unspec...|    62|
|Cardiac arrhythmi...|    54|
|Sepsis, unspecifi...|    47|
|Shock, not elsewh...|    46|
|Essential (primar...|    42|
| Alzheimer's disease|    34|
|Obesity, unspecified|    34|
|Chronic ischemic ...|    33|
|Other disorders o...|    33|
|       Other obesity|    32|
|           Emphysema|    29|
|Nontraumatic intr...|    24|
|  Respiratory arrest|    21|
+--------------------+------+
only showing top 20 rows
```

10. Number of deaths by day of the week:

```
results_day_week = spark.sql(
"""SELECT count(*) as counts,
          CASE day_of_week_of_death
          WHEN '1' THEN 'Sunday'
          WHEN '2' THEN 'Monday'
          WHEN '3' THEN 'Tuesday'
          WHEN '4' THEN 'Wednesday'
          WHEN '5' THEN 'Thursday'
          WHEN '6' THEN 'Friday'
          WHEN '7' THEN 'Saturday'
          WHEN '9' THEN 'Unknown'
          END AS day
  FROM mergedTable
  GROUP BY day
  ORDER BY counts""")
```

With output:

```
+-------+---------+
| counts|      day|
+-------+---------+
|   1033|  Unknown|
|2581659|   Sunday|
|2585712|  Tuesday|
|2589390|Wednesday|
|2595314| Thursday|
|2601045|   Monday|
|2626127|   Friday|
|2630534| Saturday|
+-------+---------+
```

11. Number of deaths by ethnicity:

```
results_deaths_skin_color = spark.sql(
"""SELECT count(*) as counts,
        CASE race_recode_3
        WHEN '1' THEN 'White'
        WHEN '2' THEN 'Races other than b&w'
        WHEN '3' THEN 'Black'
        END AS skin_color
   FROM mergedTable
   GROUP BY skin_color
   ORDER BY counts""")
```

With output:

```
+--------+--------------------+
|  counts|          skin_color|
+--------+--------------------+
|  572732|Races other than b&w|
| 2033419|               Black|
|15604663|               White|
+--------+--------------------+
```

6.2. Model's accuracy

Accuracy was chosen as the evaluation metric of the implemented **Logistic Regression** model.

Then, after training the model on the data belonging to the training set, its performance was assessed on the basis of the *Accuracy* of the predictions in the data belonging to the test set.

The result obtained from this evaluation is shown.

```
# Efficiency of Model: Evaluate model
evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction")
evaluator.evaluate(predictions)

Accuracy: 0.6650842059202722
```

7. Final remarks

In this project we have experimented with the implementation of a real **Big Data Analytics** system, whose goal is to analyse a large amount of data.

We succeeded in the intent that we had proposed, which was:

1. start with a dataset on a vertical domain such as the causes of death in the United States
2. filter the data of interest through a phase of data preprocessing
3. conduct the analysis that were of interest to us with an analytics engine which is Spark
4. save the results obtained in a database like Cassandra that lent itself to the availability and partitioning of data.

We implemented a *Peer-to-Peer* architecture based on a cluster of nodes, which allowed us to study its *fault-tolerance* and scalability properties.

From this we have come to two conclusions: the system lends itself well to a production environment where data must always be available, even if the amount of data should increase, and the system does not affect performance in case of cluster node failures as the data is replicated to all nodes.