Supercomputing for Big Data ET4310 (2016)
Assignment 2

Lorenzo Gasparini (4609905)

October 26, 2016

1 Introduction

This assignment is about the analysis of data from the *Internet Movie Database* to compute the actors that are linked to the prolific actor Kevin Bacon by six degrees of separation.

This concept is based on the theory of *six degrees of separation*, which says that every human being in the world is linked to each other through a maximum of six steps, where the friendships constitute the links between people.

In our case we analysed a list of actors from IMDb to find the ones that are linked to Kevin Bacon in 6 steps, where two actors are linked if they played a role in the same movie.

IMDb is an on-line database of information about movies, TV series and video games. The site offers a subset of its database as downloadable plain text files, including the files `actors.list.gz` and `actresses.list.gz` which contain for each actor (or actress, respectively) the list of movies he or she has played in.

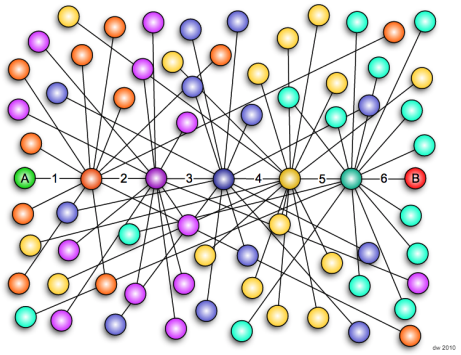

Figure 1: Six degrees of separation

The next sections are organised as follows: *Background* contains an overview of Spark's caching mechanism, SparkListener and GraphX. In *Implementation* the technical implementation details are discussed. In *Results* it's discussed whether the initial goals were achieved successfully. Finally, *Conclusion* contains a summary of the report with additional remarks on what was learnt during the assignment.

2 Background

2.1 Caching and serialization

Caching is one of the most important parts of Spark and is also what gives it a significant performance advantage over Apache Hadoop.

Normally, all the transformations in Spark are *lazy*. This means that they do not compute the results right away, but they just remember the transformations applied to the input data-set. The transformations are actually applied only when an action that requires a result triggers the execution.

This laziness is crucial to the efficiency of Spark, but it means that each transformed RDD gets recomputed every time an action is called on it. If it's known that an RDD will be used for multiple transformations, Spark can be configured to persist it in memory using the `cache` or `persist` methods. In this case Spark will store the results of the transformations in memory (or disk) so that the next time the results are used there is no need to recompute them.

There are different *storage levels* that can be used to persist RDDs, most importantly the following:

- `MEMORY_ONLY`: The RDD is stored as deserialized Java object in the JVM. Default level.
- `MEMORY_AND_DISK`: Same as above but if the RDD does not fit in memory the excess is stored on disk and retrieved when needed.
- `MEMORY_ONLY_SER`: The RDD is stored as serialized Java object. This uses less memory but is more CPU-intensive. The default serializer is the Java one, but others can be used for improved performance and space-efficiency.
- `MEMORY_AND_DISK_SER`: Same as above but the excess is stored on disk.

Moreover, Spark's cache is fault-tolerant meaning that any RDD partition is lost it will be automatically re-computed. There are also options for additional replication, like `MEMORY_ONLY_2` meaning that the partitions get persisted on two cluster nodes.

The function `cache()` is merely an alias for `persist(StorageLevel.MEMORY_ONLY)`. Different types of storage level can then be set with `persist`.

As far as serialization is concerned, the serialization library suggested by the Spark documentation is *Kryo*, which is said to provide speed improvements as much as 10x over the default Java serialization.

The choice of the serialization library can be made changing the `spark.serializer` property, for example setting the value `org.apache.spark.serializer.KryoSerializer` to use the Kryo serialization library.

2.2 Compression

Another option that Spark gives to reduce memory occupation is to compress the RDD partitions. This option is disabled by default and can be enabled by setting the `spark.rdd.compress` property to `true`. When it's enabled the serialized RDD partitions are stored compressed (this has no effect if RDDs are stored deserialized). This lowers memory usage at the cost of some additional processing power.

It is possibile to choose different compression codecs, through the `spark.io.compression.codec` property. The default compression codec is LZ4, which is focused on compression and decompression speed. Other compression codecs are LZF and Snappy.

2.3 SparkListener

SparkListener is an internal developer API that can be used to intercept events from the Spark scheduler. The API is used by Spark itself for the web UI, among the others. It can also be used to gain insight in the execution of Spark programs.

A new listener can be created inheriting the `SparkListener` class and overriding the relevant methods, such as `onJobStart`, `onStageCompleted` and others. When the event is intercepted the callbacks get called and they are provided with relevant information about the event.

The listeners in order to be enabled have to be registered using the `addSparkListener` method of the `SparkContext` class.

2.4 GraphX

For the computation of the shortest path between two actors the GraphX component was used. GraphX is Spark's API for graph computation, introduced in 2014.

The main abstraction used is the `Graph`, which is a directed multigraph. This component enables efficient graph computation, offering a rich and highly flexible API with a variety of graph algorithms.

In our case the vertices will represent the actors while the edges between them will represent a collaboration.

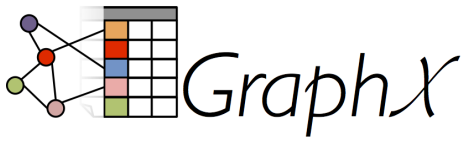

Figure 2: GraphX

3 Implementation

3.1 Shell script

The `run.sh` Shell script was modified in order to be able to choose the amount of memory used for the Spark driver via command-line arguments. This was done since development has been conducted on a machine with 32GB of RAM to speed up computations.

The syntax used to run the program is now

```
./run.sh [cpu_cores] [memory] [actors_file] [actresses_file]
```

3.2 Python script

The `rmIMDBHeaderAndFooter.py` Python script used to remove the header and the footer from the input files was ported to Python3.

3.3 Spark program

3.3.1 Input format

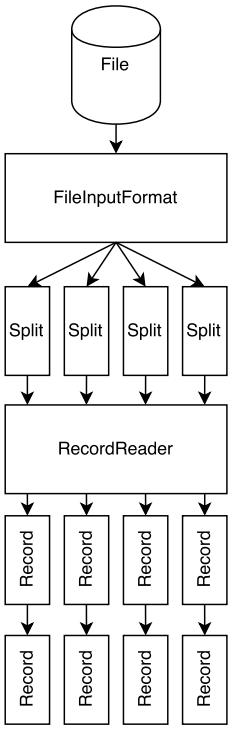

Figure 3: InputFormat diagram

Once the input files have been uncompressed and the headers and footers have been removed, some logic is needed in order to tell Spark how to read them, since unfortunately the format in which they are structured is non-standard.

The requirements state that the `newAPIHadoopFile` function has to be used to parse the files; this function is used to create a key-value RDD from a file using an arbitrary Hadoop `InputFormat`.

In our case the key-value pairs will have as the key the name of the actor and as the value the list of movies he or she has played a role in. The classes of the keys and as the values are `Text` and `ArrayWritable` respectively, which are Hadoop wrapper classes that implement the `Writable` interface for serialization.

The `InputFormat` describes the specification of the input for a Map-Reduce job. It is used for:

- Validating the input
- Splitting the input in multiple logical `InputSplits` each of which gets assigned to a mapper
- Providing a `RecordReader` which is used by the mapper to read the input records from the logical `InputSplit`

In our case, since we are dealing with files, the custom `InputFormat` that is named `IMDBActorsInputFormat` is a subclass of `FileInputFormat`. This class implements the logic for splitting the file into multiple chunks and feeding them to the mappers. Our custom input format is used only to specify the custom `RecordReader` named `IMDBActorsRecordReader`, overriding the `createRecordReader` abstract method.

Dealing with raw files and splits can be tricky. Fortunately the class `LineRecordReader` exists, which implements all of the logic needed to read files from the file system and deal with input splits. By default, as the name suggests, this class emits as a key-value pair for each line, where the position in bytes from the start of the file is the key and the value is the text of the line itself.

The default delimiter used to identify lines in the input file is the newline character `\n`. This doesn't suit our needs since it could result in an incomplete movie list for the last actor of the split, without the ability to recover the missing movies for the actor.

This is quickly solved by configuring the delimiter of the `LineRecordReader` to the double newline string (`\n \n`), which separates the actor-movies groups. This way each "line" read really is an actor-movies group, and we have the guarantee that it will never be incomplete.

The main logic of the recorder is found in the `nextKeyValue` function. Here the line record reader is used to read a "line" (actor-movies groups) from the file, which consists of multiple lines with the first containing the name of the actor. The `getCleanedMovie` function is used to obtain the name of the movie from a given line of text, if there is a valid one. For a movie to be valid it has to not start with quotes (in that case it's a TV series) and the year of release must be equal or greater than 2011.

The while loop is used since the function has to always generate a new key-value pair, and it's possible that the actor we are processing doesn't have any valid movie. In that case we read more actors until we find one that is valid.

The generated key-value pair is stored in the class variables `key` and `value` and retrieved through the methods `getCurrentKey` and `getCurrentValue`.

3.3.2 Main program

In the main program, the first step is to create the `SparkSession` setting the relevant configuration properties. Two variables are used to set the status of RDD compression and the storage level of persisted RDDs. The configuration property `spark.driver.maxResultSize`, which limits the total size of serialized results of all partitions for each Spark action, was set to `2g` since the default (`1g`) was being exceeded.

Then the custom `SparkListener` is implemented and registered intercepting the `onStageCompleted` event.

Afterwards the two files containing the list of actors/actresses and movies are loaded. A `union` is applied to them and the result is zipped with a unique ID which is used later in the graph computation. Then the ID of Kevin Bacon is obtained.

Next, a (movie, actor) RDD is created and it is joined with itself to compute the (actor1, actor2) RDD where the two actors have played a role in the same movie. Each collaboration between actor1 and actor2 is always represented with the two tuples (actor1, actor2) and (actor2, actor1). This is crucial for the usage of GraphX since it effectively removes the directionality of the edges.

This RDD is then used to create a graph in which the actors are the vertices and the collaborations are the edges. Only the IDs of the actors are used in the graph creation since the shortest path algorithm removes the attributes from the vertices so storing the name and gender there would not be useful.

Then the shortest path algorithm is ran, using as a target vertex the one of Kevin Bacon. This algorithm finds all the vertices linked to the target vertex and computes the shortest distance from each of them to the target. The result is a graph in which the vertices have an attribute representing the shortest path to the target vertex.

We then extract from the resulting graph the vertices and join them with the actors RDD to get back the name and gender information.

Afterwards, the output is printed. The movie count is found reducing by the movie name the (movie, actor) RDD.

4 Results

The program was run on a 8-core Intel® Atom™ C2750 system with 32GB of RAM, limiting the memory available to the driver to 5GB and using all the available cores.

The tests were conducted in 4 different scenarios:

- **No serialization**: RDDs persisted as deserialized Java objects. Compression has no effect in this case.
- **Java serialization**: RDDs persisted as serialized Java objects using the default Java serializer, compression disabled.
- **Kryo serialization**: RDDs persisted as serialized Java objects using the Kryo improved serializer, compression disabled.

- **Kryo + Compression**: RDDs persisted as serialized Java objects using the Kryo serializer, compression enabled.

	No serialization	Java serialization	Kryo serialization	Kryo + Compression
`maleActors`	262MB	84MB	80MB	38MB
`femaleActors`	135MB	43MB	41MB	20MB
`actors`	8MB	3MB	2MB	1MB
`actorsMovies`	602MB	183MB	137MB	65MB
`actorsDistance`	115MB	41MB	27MB	17MB
`actorsFrom1to6`	115MB	41MB	27MB	17MB
`actorsAtCurrentDistance (1)`	0MB	0MB	0MB	0MB
`actorsAtCurrentDistance (2)`	18MB	6MB	4MB	2MB
`actorsAtCurrentDistance (3)`	72MB	26MB	16MB	9MB
`actorsAtCurrentDistance (4)`	21MB	8MB	5MB	3MB
`actorsAtCurrentDistance (5)`	2MB	0MB	0MB	0MB
`actorsAtCurrentDistance (6)`	0MB	0MB	0MB	0MB
Time taken	7 mins 38 secs	8 mins 57 secs	7 mins 57 secs	7 mins 33 secs

Table 1: Memory usage and execution time in the different scenarios

Table 1 contains a summary of the memory consumption of the named RDDs and the execution times, in the different scenarios.

As expected, the memory used by the RDDs decreases from left to right. It is interesting to note that Kryo serialization is not only more memory-efficient than the Java serialization but is also an improvement from the execution time point of view.

The lowest execution time surprisingly belongs to the last scenario, and not to the first one. This can be explained by the fact that since each RDD uses less memory, more RDD partitions can be persisted in memory, leading to a speed-up.

A test was made also with RDD persistance disabled and in this case the running time was of 8 mins 15 secs.

4.1 Sample output

The following is an excerpt of the output of the program in the scenario with RDDs stored as deserialized Java objects:

```
$ ./run.sh 8 5g data/actors_noheaders.list data/actresses_noheaders.list
Driver memory: 5g
Number of cores: 8
Input files: data/actors_noheaders.list and data/actresses_noheaders.list
Total number of actors = 1321204, out of which 874456 (66.19%) are males
while 446748 (33.81%) are females.
Total number of movies = 287362

There are 1502 (0.17%) and 720 (0.16%) actresses at distance 1
There are 122012 (13.95%) and 69122 (15.47%) actresses at distance 2
There are 477143 (54.56%) and 262901 (58.85%) actresses at distance 3
```

```
There are 152779 (17.47%) and 72646 (16.26%) actresses at distance 4
There are 17066 (1.95%) and 7392 (1.65%) actresses at distance 5
There are 1931 (0.22%) and 879 (0.20%) actresses at distance 6

Total number of actors from distance 1 to 6 = 1186093, ratio = 0.8977364
Total number of male actors from distance 1 to 6 = 772433, ratio = 0.8833297
Total number of female actors from distance 1 to 6 = 413660, ratio = 0.9259359

List of male actors at distance 6:
1. AJ (I)
[...]

List of female actors at distance 6:
1. Abadou, Samira
[...]
{Time taken = 7 mins 38 secs}
```

5 Conclusion

To conclude, this assignment has given me a good insight into processing intensive Spark applications, and an overview of the GraphX framework.

The algorithm was able to compute all the distances between every actor and Kevin Bacon in a reasonable amount of time; running the algorithm with 32GB of memory available to the driver leads to a execution time near 3 minutes. Considering the program was run on a machine with a low amount of processing power, it is reasonable to assume that there are wide margins of improvement in terms of execution time just by using a more powerful machine.