COMP 47470

Big Data Programming

Project 3

Conor Murphy

20205251

01/05/2021

Section 1 Hadoop MapReduce

The source code for each question is viewable in the Hadoop folder and each file is named after the question it addresses.

Question 1

During the map phase the key is set to the movie name and the value set to one, this is repeated for each item in the data set. The reducer then aggregates each key value pair meaning that the output will show the number of actors per movie. Show below is a portion of the output. The script is viewable in q1.java

Movie	No. Actors
Across the Wide Missouri (1951)	1
Actor's Notebook: Christopher Lee (2002)	2
Adam Sandler Goes to Hell (2001)	26
Adaptation. (2002)	1
Addams Family The (1991)	1
Addams Family Values (1993)	1

Question 2

This question is identical to the first but this time an if statement is present in the mapper which checks if the actors name is equal to Peter Cushing, if it is then it is then sent to the reducer. In this instance only one entry is being sent to the reducer. The code is viewable in q2.java. The full list of actors in that movie could have been included if necessary but as the question only asked for the movie that peter was in this was all that was included.

Answer: Actor's Notebook: Christopher Lee (2002)

Question 3

This question got a little more complicated and the solution is split into three parts mapper1, reducer1 and mapper2. The first is the initial mapper which separate the input file into key value pairs of (movie, Actor). The reducer then receives these and using String concatenation appends the actors' names together for the same key, this creates an adjacency list this is then stored in a temporary file in the hdfs. Another mapper job is then started which reads the adjacency list. In this mapper the list of actors is transformed into an arraylist and the using the contains method we can check if the list contains both Cate Blanchett and Christina Ricc, if it does it is then the movies and its actors are written to the outputfile. Another Reducer could have been added but as the result is already concatenated and only one of each movie is present it was not necessary. The code below allows for the one java file to kick off multiple map and reduce functions and because of this the user only need to only run the file once. Note the job.waitForCompletion method which will ensure the first map and reduce are complete before stating the second map phase, by using multiple configurations it allows for multiple map and reduce jobs to be called from the main method. The full code is available in q3.java

```
public static void main(String[] arg0) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "job1");
    job.setJarByClass(q3.class);
```

```java
    job.setMapperClass(MyMapper1.class);
    job.setReducerClass(MyReducer1.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);
    FileInputFormat.addInputPath(job, new Path(arg0[0]));
    FileOutputFormat.setOutputPath(job, new Path(arg0[1]+"temp"));
    job.waitForCompletion(true);

    Configuration conf2 = new Configuration();
    Job job2 = Job.getInstance(conf2, "job2");
    job2.setJarByClass(q3.class);
    job2.setMapperClass(MyMapper2.class);
    job2.setMapOutputKeyClass(Text.class);
    job2.setMapOutputValueClass(Text.class);
    FileInputFormat.addInputPath(job2, new Path(arg0[1]+"temp"));
    FileOutputFormat.setOutputPath(job2, new Path(arg0[1]));
    System.exit(job2.waitForCompletion(true) ? 0 : 1);
}
```

Answer: Man Who Cried The (2000) : Christina Ricci,Cate Blanchett,

Question 4

This is a very similar approach to the question three except here we are checking if the array contains (Audrey Gelfund & John Malkovich) or (John Malkovich & Kevin Bacon) the code below shows the second map function where it checks if the arrayList contains the two specified actors. I decided to only show the actors it is looking for as the movies contained too many actors that the output became messy and unappealing, so the result was set to (actor1 -> actor2) to make it more readable. This approach was taken as it allowed for only one java file to be run which will return one output with the entire answer.

```java
public static class MyMapper2 extends Mapper<LongWritable, Text, Text,
Text> {

    protected void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
        String[] cols = value.toString().trim().split("=");
        String[] actrs = cols[1].trim().split(",");
        List<String> lst = Arrays.asList(actrs);

        if(lst.contains("Audrey Gelfund") && lst.contains("John
Malkovich"))
            context.write(new Text(cols[0] + ":"), new Text("Audrey Gelfund
-> John Malkovich"));

        if(lst.contains("John Malkovich") && lst.contains("Kevin Bacon
(I)"))
            context.write(new Text(cols[0] + ":"), new Text("John Malkovich
-> Kevin Bacon"));
    }
}
```

Answer:

Being John Malkovich (1999) : Audrey Gelfund -> John Malkovich
Queens Logic (1991) : John Malkovich -> Kevin Bacon

Question 5

This was a near identical approach to the previous question but this time the condition was checking if a movie contained Kevin Bacon AND if it has more than one actor, this guaranteed that movies only with Kevin Bacon present were not included in the output. The full code is viewable in q5.java

```
if(lst.contains("Kevin Bacon (I)") && lst.size() > 1)
    context.write(new Text(cols[0] + ":"), new Text(cols[1]));
```

Answer:

```
Murder in the First (1995)	:	Christian Slater,Kevin Bacon (I),
Queens Logic (1991)	:	John Malkovich,Kevin Bacon (I),
She's Having a Baby (1988)	:	Kevin Bacon (I),Bill Murray (I),
Wild Things (1998)	:	Bill Murray (I),Kevin Bacon (I),
```

Section 2: Graph X

The following questions will all require q1 to be run first as it sets the graph up and is used in all subsequent questions.

Question 1

Initially I planned to have the actors be the vertices and the movies be the edges but later realised that firstly the data set would need to be transformed to make this possible and also that this approach is not necessary for the questions being asked. As such I decided to implement both actors and movies as vertices and connect them with an unweighted edge making it an undirect graph, an example of the approach can be seen in figure 2.1

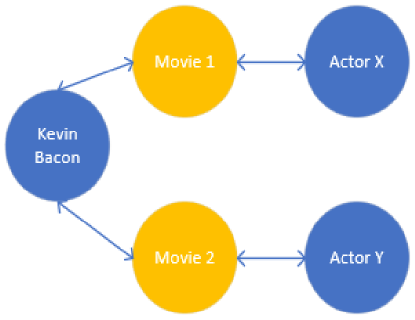

Figure 2.1

The below code shows the text file being read in from a folder called /asgn3/bacon.csv. The header row is firstly removed, each didtinct vertex is then taken form the file, this is important as we do not want two nodes for the same actor or movie in the graph, The edges are then read in, a value of one is set for the edge value but this will never be used. A default point is necessary as GraphX requires it to create the graph so is point is initiated as default. Finally the graph is created by passing the vertices, edges and starting point as parameters. The graph is now set up and can be used in the subsequent questions.

```
import org.apache.spark._
import org.apache.spark.rdd.RDD
import org.apache.spark.util.IntParam
import org.apache.spark.graphx._
import org.apache.spark.graphx.util.GraphGenerators

case class Film(actor:String, actorID:Long, movie:String, movieID:Long)
//create a film object(includes movies and actors)
def parseFilm(str: String): Film = {
    val line = str.split(",")
    Film(line(0), line(1).toLong, line(2), line(3).toLong)
}

var textRDD = sc.textFile("/asgn3/bacon.csv")
val header = textRDD.first()
textRDD = textRDD.filter(row => row != header)
val filmRDD = textRDD.map(parseFilm).cache()
val verts = filmRDD.flatMap(film => Seq((film.actorID, film.actor),(fil
m.movieID, film.movie))).distinct

val connections = filmRDD.map(film => (film.actorID,film.movieID)).dist
inct
val edges = connections.map {case (actorID, movieID) =>Edge(actorID, mo
vieID,1) }
//default vertex
val default = "default"
val graph = Graph(verts, edges, default)
```

Question 2

The number of vertices can be found using the graph.numVertices method and the number of edges found using the graph.numEdges method, the. The number of movies and actors' nodes is also shown by filtering out all values less than 999 for the actors and the movies can be found by filtering out all nodes with an id over 999. The code is shown below with the results shown as a comment under the code.

```
val numverts = graph.numVertices
//long = 1747
val numEdges = graph.numEdges
//Long = 1817

val numMovies = graph.vertices.filter{case(v) => v._1 > 999}.count
//Long = 1586

val numActors = graph.vertices.filter{case(v) => v._1 < 999}.count
//Long = 161
```

Question 3

```
val deg = graph.degrees //undirected graph so no ned for in or out
val result = deg.filter {case ((movieId, num)) => num > 2}.filter {case
 ((movieId, num)) => movieId > 999}

val filmMap = verts.map { case ((id), name) => (id -
> name) }.collect.toMap
val more_than_2_actors = result.map(x => (filmMap(x._1), x._2))
more_than_2_actors.foreach(println)
```

Firstly, the number of degrees can be found using the graph.degrees method , this is used over the inDegrees or outDegree methods as the graph is undirected. The first filters will filter out all the vertices that are connected to less than 2 other nodes and as actor nodes are only connected to movie nodes this means that if they have less than 2 degrees it means less than 2 movies. Then this list still contains actors, so all actor nodes are removed as their id will be less than 999. The list is still in the format (1005, 3) where the first number is the Id and the second is the number of actors in the film, to make this more readable I mapped this value to its corresponding name and then printed the results.

A snippet of the output is shown below, the number of movies is indicated as the second item in the result. the full list of the 19 movies can be seen by running the code above or found in q3.sc.

Answer:

```
(That's Entertainment Part II (1976),3)
(Robin Hood: Prince of Thieves (1991),3)
(Century of Cinema A (1994),5)
(That's Dancing! (1985),3)
(Star Wars: Episode VI - Return of the Jedi (1983),3)
```

Question 4

```
val kb_movies = graph.triplets.filter{triplet => triplet.srcAttr == "Ke
vin Bacon (I)"}.foreach(println)
//can use the count to find out the total number = 57
```

This question can be done by utilising triplets which contain the source, destination, and edge value (which is always 1 in this case). Due to the way the graph was set up and the manner in which it was read where every edge contained an actor and the corresponding movie they were in, this allowed for all sources but Kevin bacon to be filtered out which subsequently will return all the movies he was in. Shown below are the first three entries. The full list is viewable by running the code.

Answer

((4,Kevin Bacon (I)),(1003,Animal House (1978)),1)

((4,Kevin Bacon (I)),(1002,Air Up There The (1994)),1)

((4,Kevin Bacon (I)),(1006,Beauty Shop (2005)),1)

((4,Kevin Bacon (I)),(1004,Apollo 13 (1995)),1)

Question 5

```
val deg = graph.degrees
val result = deg.filter {case ((movieId, num)) => movieId < 999}
val filmMap = verts.map { case ((id), name) => (id -
> name) }.collect.toMap
val most_actors = result.map(x => (filmMap(x._1), x._2))
most_actors.sortBy(_._2, ascending=false).take(5).foreach(println)
```

This is similar to question 3 in that it gets the degrees of each node, as the question is asking for actors all of the movie nodes are filtered out. The result is then mapped to the actor's name as they are more readable than their ID. Finally, the name result is sorted in descending order meaning that the top results will be the largest. I am displaying the top 5 but if the top one was only desired then take(1) could be used. The result Is shown below with Christopher Lee being in the most films with an impressive 218 appearances.

Answer:

(Christopher Lee (I),218)

(Clark Gable,114)

(James Stewart (I),112)

(James Earl Jones,101)

(Cary Grant,101)

Section 3 Reflection

Before discussing the two approaches to finding the second neighbour in Hadoop and spark it is helpful to understand the problem, take the sample graph in figure 3.1 for example here the nodes E,D and G are all second neighbours from A as they are connected to nodes B and C, F would be a third neighbour.

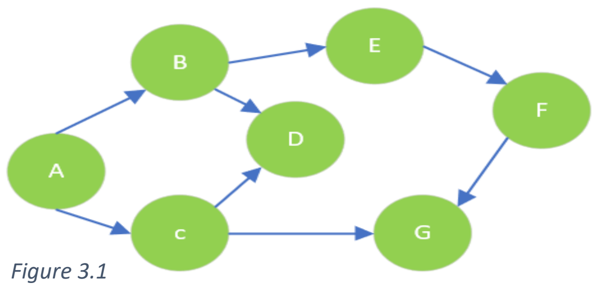

Figure 3.1

Hadoop Approach

The first step is to represent the graph in a suitable manner, I would recommend an adjacency list as opposed to an adjacency matrix as with large datasets an adjacency matrix can be very sparse. Once the adjacency list is established then the first node needs to be selected, continuing with the example in figure 3.1 this would be set to node A. In the first Map iteration the connected node values of A are passed to the reduce function which are found using the adjacency List. In the reduce function the two connected nodes of B and C can then be Considered First Neighbours. Now on the next Map Reduce iteration the nodes B and C are considered Active nodes so their connected nodes need to be found, this can once again be done using the adjacency list. On this Map iteration the map function will pass only the adjacency list values of the active nodes B and C to the reduce function and filter out all of the other values. the reduce function now receives the adjacency list values for B and C and using this their connected nodes can be found which are E,D and G. This is where the parallel nature of map reduce comes in very handy as this number of nodes could be in the hundreds so with more machines each of these nodes could be traversed to more easily. In the context of this problem the program would terminate here as these values are the second neighbours, but if the third or fourth neighbours were required this iterative approach could continue to be used, where the adjacency list is used to find the next set of connected nodes at each iteration.

One of the downsides to this approach is that nodes that have already been visited can be visited again. For example, if Node B had an edge connecting it to C then this would be considered a second neighbour through the path A->B->C which is not true as it is actually a first neighbour from A-> C. to ensure that this does not happen in the approach we can assign a flag to keep track if a node has been visited so each node will only be visited once which would solve our problem. Another problem with this approach is that fact that after each iteration the entire adjacency list will need to be read to find the next node, this is due to Hadoop not being able to store intermediate results in between jobs in memory, because of this at every iteration the list will need to be re-read, shuffled and the desired nodes emitted for every iteration making this a costly operation. This unfortunately cannot be overcome and is just one of the traits of Hadoop map reduce.

GraphX Approach

In this approach we can utilise a specific graph component of spark called GraphX. This comes with many added benefits, the biggest one is the fact it leverages the power of RDD's (Resilient distributed datasets) which allow for in memory processing. The first step would be to load the desired vertices and edges into a graph, if weighted edges are present in the context of this problem thay can be ignore, if the shortest path to a second neighbour was required this would be different. We can set A as the starting node for the graph and then using a graphX method like collectNeighbours we can generate an RDD with the Ids and attributes of the first Neighbours of A. Now we have a list of all of the first neighbours so we can iterate over this and using the original graph we can collect all of their neighbours in the same fashion we just found A's, which will return a list of all the second neighbours. It is useful to note that spark separates the static graph structure from the active vertex sets (first neighbour set B,C is an example of an active vertex set in this example) that we are generating at each iteration so this means that cached RDD partitions are used which will remove the need for costly reshuffle operations of the adjacency list like in Hadoop. These steps could also be used to generate a subgraph of only the first and second neighbours if needed. This solution will also be considerably faster than the Hadoop implementation due to the use of RDD’s meaning that the intermediate step in between the first neighbour and the second neighbour can be passed in memory and does not need to be written back to the file system, which will drastically save on the run time.

It should be noted that neither of these approaches are really necessary for second neighbour computation unless the graph is exceptionally large, a simple breath first algorithm could run a lot faster than the Hadoop implementation if there are only a few nodes to travers and using graphX for a smaller graph defeats its intended purpose of large scale distributed graphs processing and may be somewhat overkill. In the event the graph is large enough or a further degree of neighbour is needed then graphX is the more optimal solution due to Sparks use of RDD’s and in memory processing.