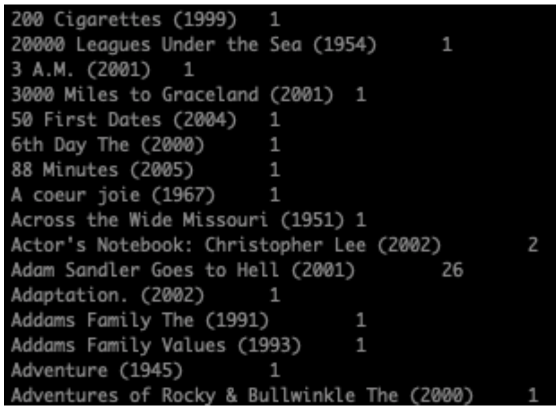
Hadoop Graph Processing

1) Create a list of the number of actors that appear in each film.

See question1/CountActors.java

I treated this question as a simple word counting problem. Here, instead of counting individual words, we are counting movie titles. The mapper splits the input using the comma as a delimiter, and then selects the value at index 2, which refers to the "to" column, containing the movie titles. The movie title then becomes the key, and it is written to the context with a value of one. The reducer then combines those key-values pairs to generate a count of how many times each movie is featured, i.e. how many in-links there are to each movie. This represents the number of actors that appear in each film.

The resulting output for this question is very long, so I've provided an example output snippet on your right, for the full result please run the provided code.



```
200 Cigarettes (1999)   1
20000 Leagues Under the Sea (1954)      1
3 A.M. (2001)   1
3000 Miles to Graceland (2001)  1
50 First Dates (2004)   1
6th Day The (2000)      1
88 Minutes (2005)       1
A coeur joie (1967)     1
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
Adventure (1945)        1
Adventures of Rocky & Bullwinkle The (2000)     1
```

Sample output for question one

Q2) Peter Cushing appears in only one film - what is it?

See: question2/PeterCushing.java
Answer: Actor's Notebook: Christopher Lee (2002)

I approached this in a fairly similar manner to the previous question. I split the the input string, using the comma as a delimiter, and verified if the value at index 0 (the "from" column") in each line of input was equal to "Peter Cushing". If it was, it was written to the context with "Peter Cushing starred in" as the key, and the movie title as the value. Given that it was known in advance that this actor starred in only one movie, it was unnecessary to give an iterable value to the reducer. The key/value pair is written directly to the context.

Q3) What film connects Cate Blanchett and Cristina Ricci? i.e. what film did they both act in.

See question3/Linked.java
Answer : Man Who Cried The (2000)

My mapper implementation is extremely similar to my implementation for question 2. The key difference is that here I write the actor and movie to the context only if the actor is Cate Blanchett or Cristina Ricci. The movie title is used as the key, and the actors name is the value. In the reducer I create a HashMap for each movie-key and put each actor-value into the HashMap. I chose a HashMap as it would mitigate any data entry errors, for instance if "Cate Blanchett", "Movie A" was entered twice in the data, it would still appear only once in the HashMap. If the size of the Hashmap is equal to two, having iterated through all values associated with the movie key, we know that both Cate Blanchett and Cristina Ricci have acted in said movie, and the title is written to the context.

Q4) What is the path between Audrey Gelfund and Kevin Bacon? This will take the form Audrey Gelfund - ? - John Malkovich - ? - Kevin Bacon

See: question4 folder
Answer: Audrey Gelfund -> Being John Malkovich (1999) -> John Malkovich-> Queens Logic (1991) -> Kevin Bacon (I)

For this question I wrote two mappers and reducers, which are run consecutively by the driver. The first mapper iterates through the dataset and writes to the context any movies in which Audrey Gelfund, John Malkovich, or Kevin Bacon have acted. The film title is used as the key, and the actor as the value. The reducer then aggregates those values, adding the actor's associated with a particular movie key to an ArrayList. If the array list has a size greater than 1, it means that two of the actors have acted in said movie and it is a link. The reducer then aggregate the actor's names and the movie's title into a single string, and writes the path to the context with "path" as a key. My second mapper and reducer takes the output of the first MapReduce task (the link from Audrey Gelfund to John Malkovich, and the link from John Malkovich to Kevin Bacon) and aggregates it together to create a single path on a single line. The driver hence takes three parameters; an input path, an output path for the first MapReduce job, and an output path for the final MapReduce job.

Q5) In this dataset, 3 actors have a bacon number of 1 - they have acted in a film with Kevin Bacon. Identify these actors.

See: question5 folder
Answer: Bill Murray, John Malkovich, and Christian Slater

For this question I wrote two mappers and one reducer. The first mapper takes the movie name as the key, and the actor as the value and writes it to the context. The reducer aggregates all the actors per movie together, into a single comma separated list. The first mapper/reducer pair has converted the data into an adjacency list representing the out-links from the movie nodes. This graph is undirected, so an in-link is equivalent to an out-link when looking at the "to"/"from" data in the csv file. The next mapper iterates through the adjacency list, and checks if the list of actors related to a movie contains Kevin Bacon. If Kevin Bacon is present, we split that list of actors into a string array. We iterate through the string array, and write each actor to the context that is not Kevin Bacon. We instead use "Kevin Bacon" as the key value. The reducer then aggregates these values together into a single, comma-separated string. As with the previous question, the driver requires three input parameters; an input path, an output path for the first MapReduce job, and a final output path for the second MapReduce job.

GraphX

Q1) Import the data and create a graph representing the data
Apologies for the small size of the font in the code section. Newlines make the spark shell throw errors when copy/pasting from a pdf.

Code:

```
import org.apache.spark._
import org.apache.spark.rdd.RDD
import org.apache.spark.graphx._

val data = spark.read.format("csv").option("header", "true").option("inferSchema", true).load("/root/bacon_mirror.csv")
val edges = data.map(row => Edge(row.getInt(1).toLong, row.getInt(3).toLong, 1))
val eRDD = sc.parallelize(edges.collect().toSeq)
val vertices = data.map(row => (row.getInt(1).toLong, row.getString(0)))
val vDD = sc.parallelize(vertices.collect().toSeq)
val graph = Graph(vDD, eRDD, "none")
```

First of all, I import the relevant packages, including the graphx functionalities. A property graph requires edges and vertices. A GraphX Edge requires the node id numbers of two connected vertices, and also the distance between them. I generate this by mapping the imported data row by row, and creating "Edge" objects, with a set edge distance of 1. I then parallelise my edge values to an RDD. Next, we must create our vertices. Vertices for GraphX require the node number and also a node name. The vertices val is created by mapping the "from no" and "from" label values from the data, and parallelising the result to an RDD. A property graph can then be created, using the generated edge and vertex RDDs, and also a default value which I set to "none".

Q2) How many nodes and edges are there in the graph?
Answer: 1747 vertices and 3634 edges
Code:
graph.numVertices
graph.numEdges

GraphX keeps track of these values for you, and are returned using the above commands.

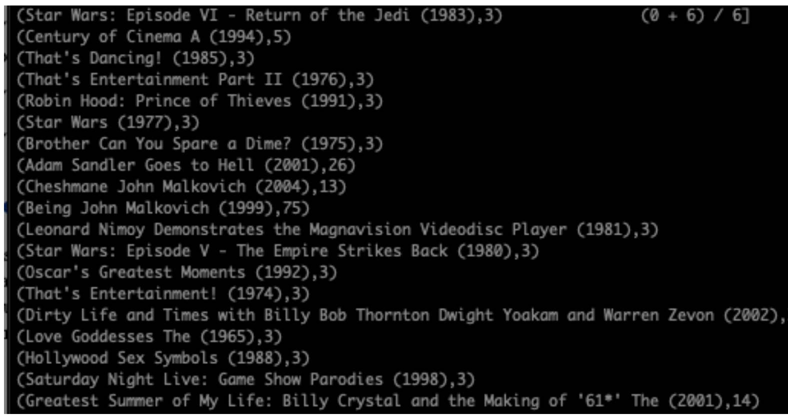
Q3) Which films star more than 2 actors in the dataset?
Answer: See screenshot.

Code:

```
val mapVertices = vertices.map { case ((_1), _2) => (_1 -> _2) }.collect.toMap

graph.inDegrees.filter(v => v._2 > 2 && v._1 >= 1000).map(v => ((mapVertices(v._1),v._2))).foreach(println)
```

I first create a mapper between the node ids and the node names. The value "mapVertices" takes the vertices value instantiated while creating the graph, and maps value 1 to value 2 of the tuple, meaning the node id is now associated with the node name. Now, if a numerical node id is given to mapVertices, it will output the associated name.



```
(Star Wars: Episode VI - Return of the Jedi (1983),3)              (0 + 6) / 6]
(Century of Cinema A (1994),5)
(That's Dancing! (1985),3)
(That's Entertainment Part II (1976),3)
(Robin Hood: Prince of Thieves (1991),3)
(Star Wars (1977),3)
(Brother Can You Spare a Dime? (1975),3)
(Adam Sandler Goes to Hell (2001),26)
(Cheshmane John Malkovich (2004),13)
(Being John Malkovich (1999),75)
(Leonard Nimoy Demonstrates the Magnavision Videodisc Player (1981),3)
(Star Wars: Episode V - The Empire Strikes Back (1980),3)
(Oscar's Greatest Moments (1992),3)
(That's Entertainment! (1974),3)
(Dirty Life and Times with Billy Bob Thornton Dwight Yoakam and Warren Zevon (2002),
(Love Goddesses The (1965),3)
(Hollywood Sex Symbols (1988),3)
(Saturday Night Live: Game Show Parodies (1998),3)
(Greatest Summer of My Life: Billy Crystal and the Making of '61*' The (2001),14)
```

Question 3 output

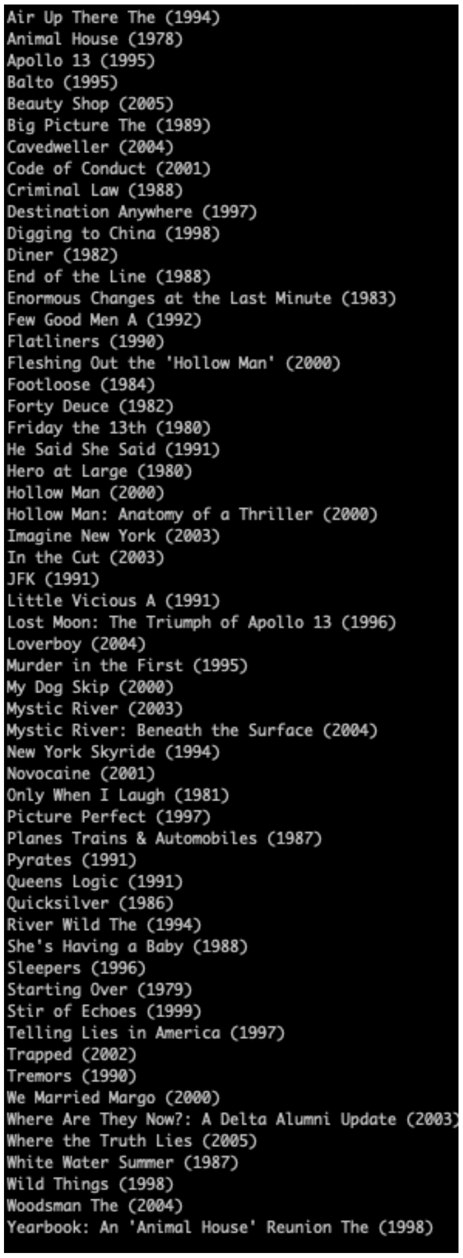
This is convenient for the readability of the results of the second line of code. Graph.inDegrees contains the in-link information for every node in the graph. This is information generated and stored automatically by GraphX when a graph is created. It is stored as a series of tuples where the first value is the node id and the second value is the number of in-degrees of that node. Given that this is an undirected graph, in-degrees and out-degrees are equivalent to each other.
We are filtering the inDegrees data, so that the second element of the tuple (the in-degrees) must be larger than 2, and the first element of the tuple (the node id) is equal to or larger than 1000. This is because all actors have a node id smaller than 1000 and all movies have a node id larger than or equal to 1000. The filtered result is then mapped to a new tuple, where we use our previously defined "mapVertices" mapper to convert the node id to the node name. For clarity and debugging purposes, I also chose to display the number of in-links beside the movie title.

Q4) What films has Kevin Bacon starred in?

Answer: Please see screenshot on the next page.

Code:
```
val kev_id = vertices.filter(v => v._2 == "Kevin Bacon (I)").map(v => v._1).take(1)(0)
val kev = graph.collectNeighbors(EdgeDirection.Out).lookup(kev_id).toArray
kev(0).map(e => e._2).foreach(println)
```

First of all, I extract Kevin Bacon's node id from the vertices object created when reading in the data. "Vertcies" contains tuples in the form (node_id, node_name), so by filtering based on the second value of the tuple, we can extract the associated node id. The resulting value is a dataset, so we use "take" to collect the first entry, which gives us an array containing the node_id Long value, which we extract by selecting index 0
Having extracted Kevin Bacon's node_id, we can use the inbuilt function "collectNeighbors". CollectNeighbors takes an edge direction (I have chosen "Out") and aggregates information about adjacent nodes in the graph. We can then use the "lookup" feature to get information about a specific node's neighbours. In this case, we use Kevin Bacon's node id to retrieve its' neighbours, which are returned in the format (node_id, node_name), from which we can map and print each node name.



```
Air Up There The (1994)
Animal House (1978)
Apollo 13 (1995)
Balto (1995)
Beauty Shop (2005)
Big Picture The (1989)
Cavedweller (2004)
Code of Conduct (2001)
Criminal Law (1988)
Destination Anywhere (1997)
Digging to China (1998)
Diner (1982)
End of the Line (1988)
Enormous Changes at the Last Minute (1983)
Few Good Men A (1992)
Flatliners (1990)
Fleshing Out the 'Hollow Man' (2000)
Footloose (1984)
Forty Deuce (1982)
Friday the 13th (1980)
He Said She Said (1991)
Hero at Large (1980)
Hollow Man (2000)
Hollow Man: Anatomy of a Thriller (2000)
Imagine New York (2003)
In the Cut (2003)
JFK (1991)
Little Vicious A (1991)
Lost Moon: The Triumph of Apollo 13 (1996)
Loverboy (2004)
Murder in the First (1995)
My Dog Skip (2000)
Mystic River (2003)
Mystic River: Beneath the Surface (2004)
New York Skyride (1994)
Novocaine (2001)
Only When I Laugh (1981)
Picture Perfect (1997)
Planes Trains & Automobiles (1987)
Pyrates (1991)
Queens Logic (1991)
Quicksilver (1986)
River Wild The (1994)
She's Having a Baby (1988)
Sleepers (1996)
Starting Over (1979)
Stir of Echoes (1999)
Telling Lies in America (1997)
Trapped (2002)
Tremors (1990)
We Married Margo (2000)
Where Are They Now?: A Delta Alumni Update (2003
Where the Truth Lies (2005)
White Water Summer (1987)
Wild Things (1998)
Woodsman The (2004)
Yearbook: An 'Animal House' Reunion The (1998)
```

Question 4 output

Q5) Which actor has starred in the most films?

Answer: Christopher Lee (I)

Code:

mapVertices(graph.degrees.reduce((a,b) => if (a._2 > b._2) a else b)._1)

The code in brackets was found in the GraphX documentation as an example function to run on a graph. The reduce function compares the degrees (i.e. out or in links) of each node in the graph and selects the node id with the largest number of degrees (connections). The resulting node id from the reduce function is fed to our mapVertices function which returns the name associated with said id.

<u>Part 3 - Reflection </u>- How would one use MapReduce and GraphX to generate a list of second neighbours (i.e node C from A in A-B-C)?

Based on the given example, we are looking to find the second neighbours of a single node, as opposed to all nodes in the graph. The algorithm that seems the most appropriate for this context is breadth first search, or BFS. The BFS algorithm is an iterative algorithm where one first visits all the neighbours of a given node, marks said neighbours as visited, and then visits all the neighbours of those neighbours, i.e. second degree neighbours. It then visits third degree neighbours, fourth degree neighbours etc. The key difference here between the given scenario and a "normal" BFS is that our stopping condition will be reaching layer two of the neighbouring nodes, as opposed to finding a specific node or traversing the whole graph.

In GraphX, in order to construct a graph, we need node ids, node labels and edge values for those nodes. Using this data, we must construct an edge RDD and a vertex RDD which we can then use to create a property graph. Once a property graph has been constructed, a number of graph processing tools are readily available to the user. One of these is "CollectNeighborIds". CollectNeighborIds takes an edge direction as a parameter and then outputs a Vertex RDD. The edge direction parameter dictates whether we look at out links, in links, or either. In this case, we would be looking at out links from a source node to its' neighbours. The outputted Vertex RDD resembles a kind of adjacency matrix; the first value of each row is the parent node id, and the second value is an array of all the neighbouring nodes. If we wished to generalise this solution to find any n layer of neighbours, we would need a loop to iterate through the process n times, while keeping track of which nodes have already been visited. But, given the simplification of the problem, the solution too can be simplified. I would use the "lookup" feature to lookup the neighbours of our source node from the Vertex RDD created by "CollectNeighborIds". I would then extract the array of neighbours, and iterate through it, looking up the neighbours of each "first" neighbour, and saving the results to a Scala mutable set. The set datas tructure is appropriate as sets do not retain duplicate values so there is no need to verify whether we have already visited a node or not. If we were to extend this proposed solution for generalised graph traversal, we would need to keep track of visited nodes in order to avoid getting trapped in cycles, which could cause an infinite loop in our algorithm.

MapReduce has no equivalent inbuilt graph processing functions, so instead we will need to write a mapper and reducer to fit the purpose. First of all, our data will need to be represented as an adjacency list. Given that we are simply looking for second degree neighbours, and not a shortest path or equivalent, we do not need to encode edge distances into the adjacency list. Initially all nodes will have their "degree distance" set to infinity in the adjacency list, except for the starting node which will have a distance value of 0. We would then need to write a mapper that iterates through the adjacency list, ignoring nodes with a distance of infinity. When this mapper finds a node with a distance of less than infinity, it should emit each neighbour of the node, and the parent degree value + 1. The mapper will also need to output the overall current state of the graph. The reducer will then compare the current distance value of a node to the newly received distance value, and update the adjacency graph with the minimum value. Normally this would be an iterative process with repeated map reduce calls until a stopping condition is reached (for instance the minimum distances no longer change), but in this context we only need to run the mapper and reducer twice. The first call to MapReduce will set the distance values of the source node's immediate neighbours to one, and with the second call the neighbours of the neighbours will have their distances set to two. From there, we can easily read which nodes are the 2nd degree neighbours, but a further map reduce task could be written to extract all nodes with a distance value of two and write them to the context.

Both GraphX and MapReduce require that we preprocess our data in a particular manner but overall I find GraphX much easier to use in the context. MapReduce is not well-suited to graph processing, as it must read and write the current state of the graph to disk for each Map/Reduce cycle. This is slow and computationally costly as compared to the graph-specific distributed tools that GraphX provides. Additionally, given that there is no interactive shell for MapReduce, writing and testing MapReduce jobs can prove tricky. A strategy to speed up this process would be to develop and test your jobs on a smaller toy dataset, but this increases your risk of overlooking data-specific bugs.