Supercomputing for Big Data ET4310 (2016)

Assignment 2

Ajaya Adhikari (2627199)

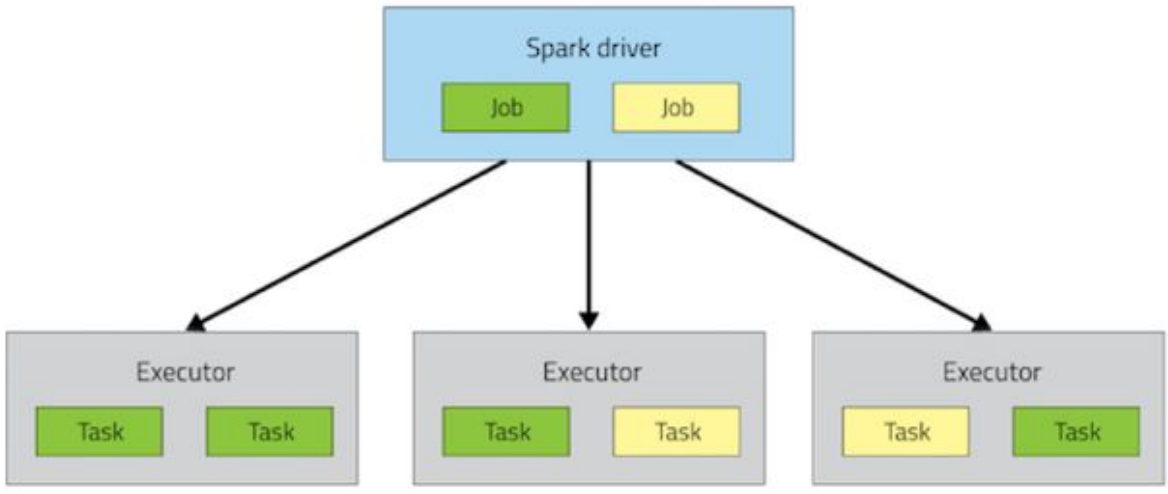
Introduction

This assignment provides a complex big data problem to be solved using Spark. The task is to compute the actors and actresses from distance one up to six degrees of separation to Kevin Bacon. For example an actor is at distance one from Kevin Bacon, if this actor has played in a movie together with him. And the co stars of this actor in another movie are at distance two from Kevin Bacon. This was a very interesting assignment, because the input was big, and you have to be very efficient and think about memory usage. This report is structured as follows: Introduction, Background, Implementation, Results, Conclusion and References.

Background

Spark execution

A Spark application consists of a driver process which is in charge of running the user's main function and executing various processes (executor) scattered across nodes on the cluster. The driver process is in charge of the high-level control flow of work, and delegates work to the executor processes. [1]



All transformations in Spark are lazy. They do not compute their results immediately. They just remember the transformations applied to some base data dataset. The transformations are only computed when an action requires a result to be returned to the driver program. Each transformed RDD has to be recomputed every time an action is run on it. If a rdd is used for multiple actions then you can persist the RDD in memory. This will enable faster access the next you query it. [1]

Jobs form the top of the execution hierarchy. Invoking an action inside a Spark application triggers the launch of a Spark job. Spark examines the graph of RDDs on which that action depends and formulates an execution plan. The plan starts from the first RDD which does not depend on another RDD. The execution plan assembles the job's transformation into stages. Each stage contains a sequence of transformations that can be completed without shuffling the full data. [2]

SparkListener

SparkListener is a class that listens to the execution events from Spark's DAGScheduler. DAGScheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling. A sparkListener can find out about the health of the Spark applications. I use spark listener to get the information about the RDDs after the completion of each stage. The RDD name, disk size, memory size, number of partitions and number of cached partitions are written to file.

Implementation

InputFormat and RecordReader

When the function textFile("file_x") is called on the a SparkContext object, a RDD is returned with the lines of file_x as records. We do not want this, in our case. We want <actor, List<movies>> as a record. To do this I defined a custom InputFormat class with a custom RecordReader class. I took the source code of Hadoop KeyValueTextInputFormat and KeyValueTextLineRecordReader as template to write my custom classes. The RecordReader filters out the tv-series and the movies created before 2011. It also removes the actors who have not played in movies after 2011. This is more efficient than filtering them out afterwards, because we would have to go through all the records again. The record reader sets the actor as key and its list of movies as value. The custom type of the InputFormat is provided to the newAPIHadoopFile function. In this way Spark automatically retrieves the RDDs with the records in the format we want.

Algorithm

I followed the method suggested on the assignment to compute the results. First a RDD of type <actor, List<movies>> is created by using the newAPIHadoopFile function as mentioned above. From that RDD anothor RDD is generated with actors and the movies they acted in (<actor, movie>). Next an RDD for actors and their collaborating actors (<actor, actor>) is created. This rdd is reduced into another RDD of type <actor, list<actor>>, where the key is an actor and the value a list of the collaborating actors. A new RDD is created describing actors and their distances to Kevin Bacon (<actor, distance>), where distance=100, except for Kevin Bacon for whom distance is 0.

The previous RDDs are joined into a new one showing the collaborating actors with a given actor at a specific distance (<actor, <distance, list<actor>>>). This list is used to generate an RDD of type <actor, distance+1>. This list is than further reduced by taking the minimum distance of each actor (<actor, minDistance>). This step is done 6 times to identify all actors with distance 1 to 6 from Kevin Bacon. For example, after the first iteration, Bacon will have a distance of 0 and every actor who worked with him would be at distance 1, while the rest of the actors would be at distance of infinite. This connectivity would spread in the next iterations.

Efficiency

If a RDD is used more than once, it is persisted. This is more efficient because it avoids recomputation. If a RDD is not needed anymore, it is unpersisted. This will release the memory resources of that RDD and can be used for other computation.

Results

I was able to generate an output close to the sample output. The number were not exactly the same but the percentages were very close.

I ran my code on the Kova machine which has 64 cores available. When I ran my tests there was 132 GB RAM available. The following are the results

- Without compressed RDDs
    - Execution time: 26 minutes
    - Memory consumptions of the cached RDDs:
        - Male Actors: 85MB
        - Female Actors: 45 MB
        - (Actor, movie): 82 MB
        - (actor, Set(collaborating actors)): 1506 MB
- With Compressed RDDs
    - Execution time: 30 minutes
    - Memory consumptions of the cached RDDs:
        - Male Actors: 44MB
        - Female Actors: 24 MB
        - (Actor, movie): 49 MB
        - (actor, Set(collaborating actors)): 1233 MB

The compressed RDDs take less space but the execution time is more compared to the task without cached RDDs.

Conclusion

This was a very interesting assignment. I learned how to define a custom InputFormat and RecordReader. This was especially challenging because there was not so much examples available, to do the required complex record reading. I learned more about the possible power of the rdd transformations, how you can do complex computations starting from simple building blocks.

References

1. http://spark.apache.org/docs/latest/programming-guide.html

2. http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/

3. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-dagscheduler.html