NTNU

TDT4225 Store, distribuerte datamengder

Assignment #4

Øving 4

Medvirkende:
Brede Yabo Kristensen

March 14, 2019

NTNU
Kunnskap for en bedre verden

1 Setup

To run the tasks, write: spark-submit task_x.py

2 RDD API Tasks

The primary focus of using the spark functions is to only transform the RDD before the final execution, which is saving it a file. This reduces computational requirement for the program. This means that we should avoid using parrallelize() or take() in the middle of the program at all cost.

2.1 Task 1

The files we are reading are rows with columns seperated by , .

I first split each column using , as a deliminator. This is done by using the map() function as it applies a function to all of the rows. The result is yet another RDD. When splitting the rows we get an array for each row, we then want the fourth column as it is the genre column. We could of used flatmap(), but we want to an rdd that has the same number of rows.

We then use the function distinct() to remove all duplicate from the rdd. Finally we count the number of rows using the count() function, which returns number.

result: 38

2.2 Task 2

We start the same way as last time, only now we get the fifth column from the artist.csv file (year). We also make sure that the rows are converted to integers as we will compare them later. We then reduce the rdd using the reduce() function. We want to return the oldest artist and do this by comparing all the rows.

Finally we print the oldest artist birth date.

Result: 1955

2.3 Task 3

Like before, we split the lines but now as a tuple (country, count), We use reduceByKey() which executes the function for each rows that have the same key, in our case, we count each time and add it to the counter in the tuple. This leaves us with the number of artist for each country.

The task specifies that we sort the total number of artist for each country descending and artist alphabetically for countries with the same number of artists. One thing to note when using hadoop/spark is that the data is split into different partitions. So we can't call sortByKey two times for each key and value, because this function shuffles the last order, therefor not preserving the previous order. We begin by sorting by countries as normal, then we map the tuples into ((count,country),country) so that we keep the country ordering when inserting it back to the normal tuple (country, count). We also specify in this sort that we want to sort only count and nothing else, or else it will also sort countries alphabetically Descending, which is not what we want.

The result is finally outputted to result_3 with coalesce, shuffle turned off to preserve the order. Read more about why coalesce is chosen in the last subsection.

2.4 Task 4

We begin by mapping out artist_id column and keeping a number 1 in the tupple from the album file. The reason for doing this is because we know that there are duplicates in artist_id because some artist have several albums. We then reduceByKey() which executes the function inside whenever we have the same artist_id. The function basically counts each time for each same artist_id and adds it to the last element in the tuple.

We then sort artist_id descending and do the same for count only now we use the same technique as we did with task 3 to preserve the previous sort order.

2.5 Task 5

In this task we create a tupple with out of two colums, (genre, number_of_sales). The sorting is done the same as task 4.

2.6 Task 6

In this task we use mapValues for mapping out the average score for the albums. This is because we only want to map out the values and is much.

This also doesn't repartition the partitions, though it's not a problem here anyway.

We then sort the keys descending like we've done previously in task 4

We then want to get the rop 10 rows without actually performing an action or outputting anything other than an RDD. This is because RDD transformations are lazy evaluated and will only be executed when it's actually needed. We therefor use zipWithIndex() to preserve the RDD and improve performance as compared to using take(). coalesce shuffle has been set to true because there is no sorting.

2.7 Task 7

We start by reading from the results of task 6 and mapping out the data.

We then start mapping out artist_id and album_id from albums.csv. We do the same with artist_id and country from artists. We then join both the rdd's on artist_id and then finally join the result with top_10_albums. A Join includes shuffeling the data between the partitions, but since the task doesn't require any order, it's fine.

2.8 Task 8

Here we do all the joining before the sorting actually takes place, becuase joining results in shuffeling of the data. The sorting is pretty straight forward as we only need to sort on names since the value for mtv score is 5 here.

2.9 Task 9

We start by mapping out (artist_id, mtv) and (artist_id, mtv) and then join them into (artist_id, total_mtv_critic, count) The last two elements will be used in calclulating the average for each artists. This could have been done as one RDD to be more effective and only calling reduceByKey() once. We then find all norwegian artists case insensitive. We then have to sort the data alphabetically and by score (descending).

2.10 Task 10

Here we first create schemes for both artists and albums csv and define their types. We then use standard sql queries too retreive our data for each task.

count() counts adds up the number of columns in the columns, distinct() removes any redundancy, min() gets the smallest integer from the columns, Max gets the maximum number from the columns.

1. a) 50000
2. b) 100000
3. c) 38
4. d) 249
5. e) 2000
6. f) 2019
7. g) 1955
8. h) 2000

2.11 Coalesce vs repartition

Coalesce is being used with shuffle being false. This is because it preserves the ordering when sorting the data into one partition. Repartition shuffles the data when it combines it into one partition. Repartition calls Coalesce with shuffle as true, which will not preserve the sort order.