March 2022

DATABASE MANAGEMENT

CCF: Fast and Scalable Connected Component Computation in MapReduce

CHAIX Lucas, MELLERIO Antoine, YACINE Kenza

1. Introduction

The aim of the project is to implement in Spark a MapReduce algorithm dedicated to finding connected components within a graph on large scale data in an efficient way. The research paper describing the algorithm is the following: *CCF: Fast and Scalable Connected Component Computation in MapReduce* (Hakan Kardes, Siddharth Agrawal, Xin Wang, and Ang Sun).

The pseudo-code is detailed as follows:

CCF-Iterate

```
map(key, value)
  emit(key, value)
  emit(value, key)

reduce(key, < iterable > values)
  min ← key
  for each (value ∈ values)
    if(value < min)
      min ← value
    valueList.add(value)
  if(min < key)
    emit(key, min)
    for each (value ∈ valueList)
      if(min ≠ value)
        Counter.NewPair.increment(1)
        emit(value, min)
```

CCF-Iterate (w. secondary sorting)

```
map(key, value)
  emit(key, value)
  emit(value, key)

reduce(key, < iterable > values)
  minValue ← values.next()
  if(minValue < key)
    emit(key, minValue)
    for each (value ∈ values)
      Counter.NewPair.increment(1)
      emit(value, minValue)
```

CCF-Dedup

```
map(key, value)
  temp.entity1 ← key
  temp.entity2 ← value
  emit(temp, null)

reduce(key, < iterable > values)
  emit(key.entity1, key.entity2)
```

We elaborated our solution as close to the pseudo-code as possible to make sure that we would not lose scalability nor efficiency by adding unnecessary steps for instance.

We are going to test the computation time varying three features :

- the type of data objects: RDD and Dataframe
- the size of the graph: number of edges in {5, 88 234, 2 420 766, 5 105 039}
- the resources used for the computing: 2 cores on DataBricks and 2 slave nodes on Google Cloud
- the implementation language: Python and Scala

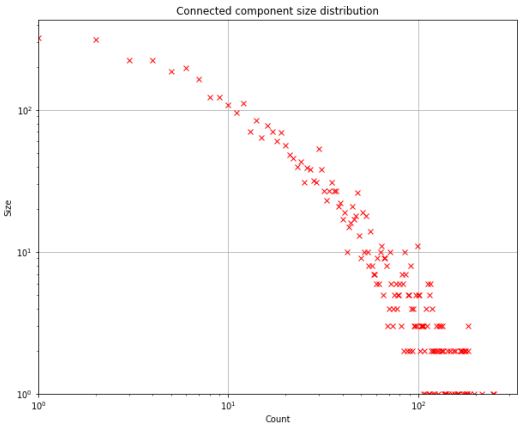
Let's first have a closer look at the four different-sized graphs chosen to conduct this work . They are imported from the following sources (SNAP Standford website).

- a micro and hand-crafted graph of 5 edges :
  [(1,2), (1,3), (4,5), (5,6), (4,6)]
- a small graph based from Facebook of 88 234 edges :
  https://snap.stanford.edu/data/egonets-Facebook.html

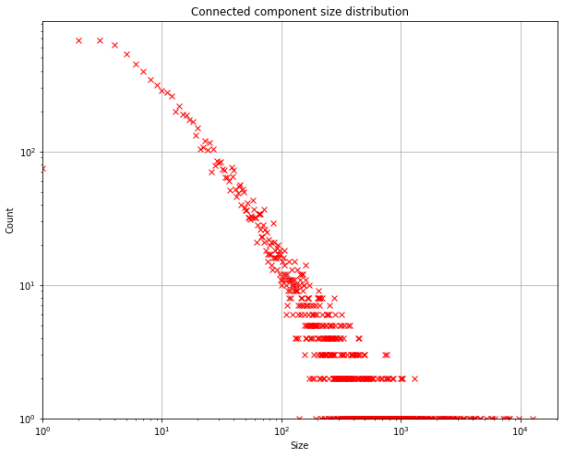
- a medium graph from Twitter of 2 2420 766 edges : https://snap.stanford.edu/data/egonets-Twitter.html
- a large graph from Google of 5 105 039 edges: https://snap.stanford.edu/data/web-Google.html

A graph's complexity is not only represented by its number of edges, but also by the number of components and their sizes. We hence plot for each graph the number of components of each size.

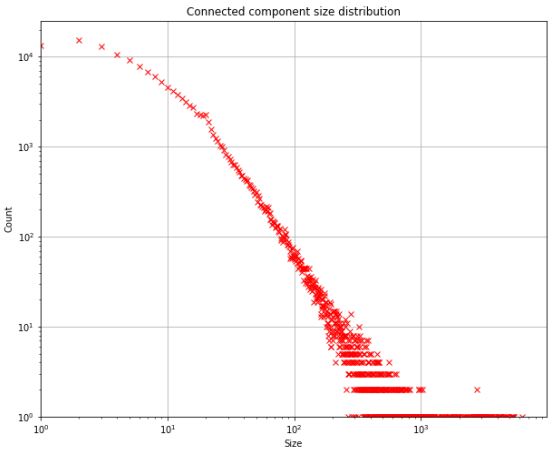
- Facebook graph



- Twitter graph



- Google graph



We can observe that even if the number of nodes and edges increases a lot, the distributions of the components' sizes are similar between each graph which makes the comparison possible.

N.B. : For convenience and better readability, all the code we present in the following sections can be found in the appendix.

2. Python implementation

2.1. Algorithm

2.1.1. Implementation on RDDs

The algorithm we built using RDDs is constructed as a spark function ***iterate_CCF*** calling a function ***iterate*** that we coded in python to follow the structure of the pseudo-code. Both the Map phase and the Reduce phase are conducted within the function ***iterate_CCF***.

To build ***iterate_CCF,*** we iterated on a small graph that we manually created and then applied it on graphs of increasing sizes mostly from social media, both on Databricks and Google Cloud in order to analyze its scalability.

COMMENTS

CCF_iterate:

Map phase:

As the pseudo-code suggests, we started by emitting both key-value pairs (k,v) and value-key pairs (v,k) to have a comprehensive list of connected nodes regardless of the direction of the edges. Then, we grouped pairs by key, in order to create for each node the list of nodes that are linked to it.

Reduce phase:

After that, we performed the reduce phase using the ***iterate*** function defined earlier in the code. It is important that we apply the ***iterate*** (or ***iterate_secondary_sorting)*** function to pairs of the form (k, [list of values]), because the Reduce phase requires going over all the values of the list to emit the right pairs as output.

We proceeded exactly the same way for the ***CCG_iterate_secondary_sorting*** function. The difference is that we need to implement a customer partitioning function to pass the values to the reducer in a sorted way.

CCF_Dedup:

Finally, we performed the final step of the algorithm described in the research paper to remove duplicates. To do so, we transformed each pair into a key: each (k,v) pair that was output from the CCF_iterate phase is turned into a ((k,v),0) pair. To remove the duplicates, we just have to reduce by key using the sum operation, and then emit only the key to recover the list of unique (k,v) pairs.

2.1.2. Implementation on DataFrames

The algorithm we built using DataFrames is constructed as a spark function ***algo*** calling a function ***iterate*** that we coded in python to follow the structure of the pseudo-code. Both the Map phase (***iterate_map_df)*** and the Reduce (***iterate_reduce_df)*** phase are conducted within the function ***iterate_CCF***.

To build ***algo,*** we used what we already had thanks to our work on RDDs, and iterated on a small graph that we manually created and then applied it on graphs of

increasing sizes mostly from social media, both on Databricks and Google Cloud in order to analyze its scalability.

COMMENTS

Map phase (***iterate_map_df)***:

As we did with RDDs, we started by emitting both key-value pairs (k,v) and value-key pairs (v,k) to have a comprehensive list of connected nodes regardless of the direction of the edges. Then, we grouped pairs by key, in order to create for each node the list of nodes that are linked to it. Note how readable this phase is when using DataFrames.

Reduce phase (***iterate_reduce_df)***:

After that, we performed the reduce phase following the indications of the pseudo-code step by step, thanks to the creation of new columns new_column_1 and new_column_2 that contain the information needed to perform our reducing process.

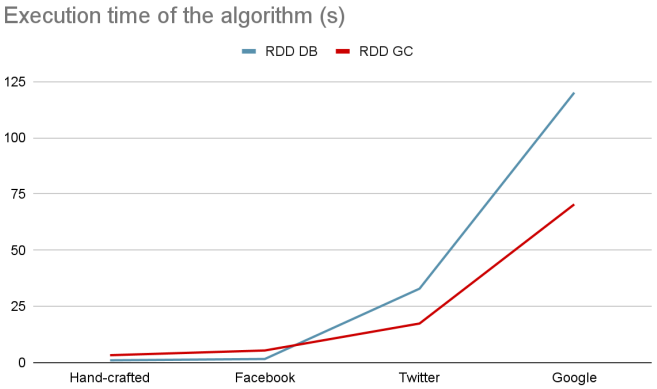
Finally, ***algo***, as described in the research paper, apply the two previous phases in order to get the result needed.

2.2. Experimental analysis

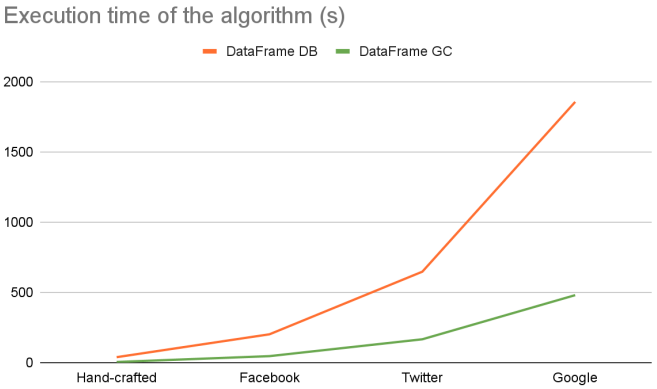
We now compute the processing of the algorithm for each graph and for each setting and store the execution time in the following table.

	RDD		**DataFrame**	
Temps d'exécution (s.) calculation+printing	Databricks Community 2 cores	Google Cloud 2 slaves nodes	Databricks Community 2 cores	Google Cloud 2 slaves nodes
Hand-crafted graph 5 edges	0.1+0.9	3.3	10.0+28.8	4.7
Facebook 88 234 edges	0.1+1.5	5.4	37.3+164.4	46.2
Twitter 2 420 766 edges	0.2+32.7	17.4	339.5+307.8	166.2
Google 5 105 039 edges	0.1+120.0	70.3	1256.4+599.4	480.3

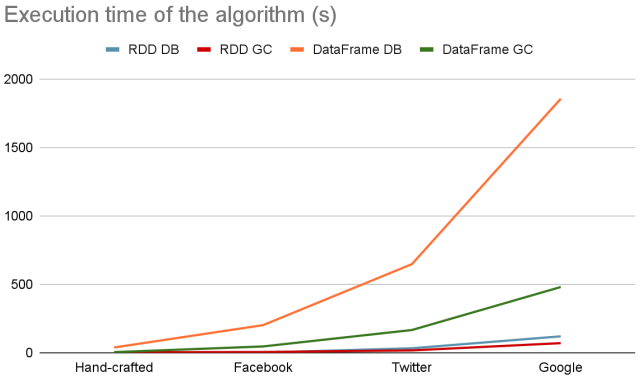
Let's plot the results to observe the configuration's influence on the execution time.



If the computation times between the Databricks settings and the GoogleCloud settings are rather similar for small graphs as RDDs, the second one appears to be much more efficient for larger graphs as RDDs.



Considering Dataframes, even though the gap appears faster than using RDDs, the result is similar : the larger the graph is, the more efficient GoogleCloud's configuration gets compared with Databricks'.
We can now compare the two data objects' influence on the computation time.



As we can see, using RDDs is at least 15 times more efficient than using Dataframe on the databricks settings.
Even though it is less obvious, the observation is the same with GoogleCloud's settings that is at least 1,4 times more efficient with RDDs.

COMMENTS

We acknowledge that our code contains a certain number of bottlenecks, but it also has some interesting advantages.

The main bottleneck is the ***groupByKey()*** function used after the mapping of the pairs. It requires performing a shuffle operation which is computationally costly. However, this step cannot be cannot be avoided since we need to generate for all nodes the list of nodes linked to it in order to find the minimum among them

The second bottleneck is the ***for loop*** used to find the minimum of the values during the reduce phase. As the number of nodes in the graph increases, this step is increasingly costly since we have to go over the whole list to find the minimum. Additionally, during this reduce phase, we also create a list of the values ***valueList*** that can use a lot of RAM memory.

On the other hand, we have also avoided some bottlenecks by hand coding some functions in order to be more efficient memory-wise. We did not use the built-in min function to find the minimum within the list of values associated with each key. Instead, we coded a function comparing values one by one and keeping in memory only the lowest one to prevent the algorithm from putting the entire list in the RAM.

3. Scala implementation

3.1. Algorithm

We built our Scala algorithm using RDDs as they gave us faster results when implementing our two Python algorithms. Our code is constructed as a spark class ***Count_Process*** including two functions ***iterate_1*** and ***iterate_2*** that we coded in scala to follow the structure of the first naive sorting and the secondary sorting, and another function ***process*** calling both functions to implement the process described in the reference paper. Here the Map phase and the Reduce phase are conducted within those three functions.

To build ***Count_Process,*** we used what we already had thanks to our work on RDDs and DFs in Python (taking into account the slowness of DFs for example), and we also iterated on a small graph that we manually created before applying it on graphs of increasing sizes mostly from social media, both on Databricks and Google Cloud in order to analyze its scalability.

COMMENTS

Count_Process

We opted for including all of our code into a class. Classes in Scala are blueprints for creating objects. They can contain methods, values, variables, types, objects, traits, etc. Here our class contains 3 functions (iterate_1, iterate_2, process) and one variable new_pairs.

Iterate_1:

We once again started by emitting both key-value pairs (k,v) and value-key pairs (v,k) to have a comprehensive list of connected nodes regardless of the direction of the edges. Then, we grouped pairs by key, in order to create for each node the list of nodes that are linked to it thanks to *.GroupByKey()* method.

After that, we performed the reduce phase using a *.flatMap()*. And it is again important to have pairs of the form (k, [list of values]), because the Reduce phase requires going over all the values of the list to emit the right pairs as output.

We proceeded exactly the same way for the ***iterate_2*** function. The difference is that we need to implement a customer partitioning function to pass the values to the reducer in a sorted way.

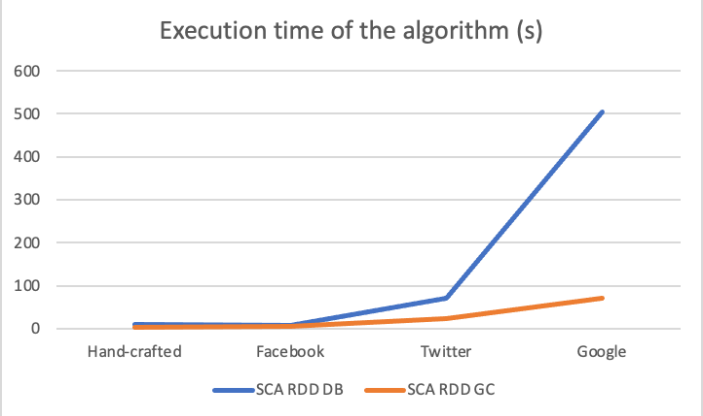
Process:

Finally, we created a final function for the final step of the algorithm described in the research paper to apply our two iterative processes described just above and remove duplicates as explained as well in our Python code for RDDs.

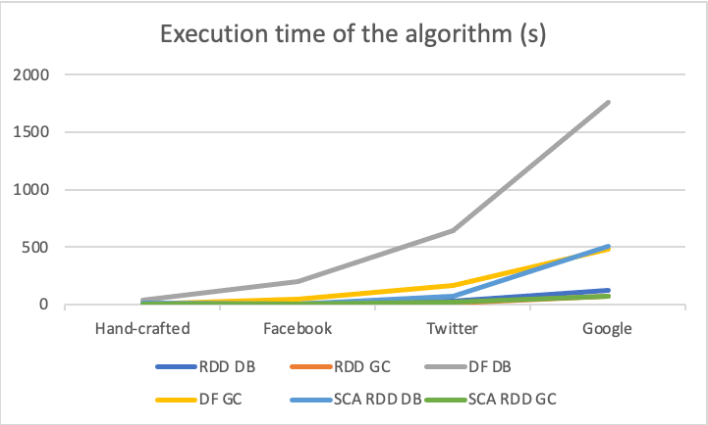
3.2. Experimental analysis

	RDD	
Temps d'exécution (s.) calculation+printing	Databricks Community 2 cores	Google Cloud 2 slaves nodes
Manual graph: **5 edges**	6.9 sec 2.5s (secondary sorting)	3.4s
Facebook: **88 234 edges**	3.7 s 3.4s	4.6s
Twitter: **2 420 766 edges**	37.7 s 35.8 s	22.8s
Google: **5 105 039 edges**	4.22 min 4.02 min	70.9s

Let's plot once again our results to observe the configuration's influence on the execution time.



If the computation times between the Databricks settings and the GoogleCloud settings are rather similar for small graphs as RDDs, the second one appears to be much more efficient for larger graphs as RDDs. Note that this difference is even larger in Scala than in Python.



As we can see, using RDDs is at least 15 times more efficient than using Dataframe on the databricks settings.

Even though it is less obvious, the observation is the same with GoogleCloud's settings that is at least 1,4 times more efficient with RDDs. Finally, using Scala or Python with RDDs on Google Cloud does not matter very much as the results are almost identical.

COMMENTS

Once again, our code contains a bottleneck as the ***groupByKey()*** function is used after the mapping of the pairs. Again, it requires performing a shuffle operation which is computationally costly. However, this step cannot be avoided since we need to generate for all nodes the list of nodes linked to it in order to find the minimum among them.

We also have a ***while loop*** used to find the minimum of the values in our process function. Once again, as the number of nodes in the graph increases, this step is increasingly costly since we have to go over the whole list to find the minimum. Additionally, during this reduce phase, we also create a third RDD (***rdd3***).

4. Conclusion

As said in the introduction of this paper, the aim of our project was to find an efficient way to implement in Spark a MapReduce algorithm dedicated to finding connected components within a graph on large scale data. To do so, we implemented the pseudo-code given in the reference paper in Python using RDDs and then using DataFrames, and in Scala using RDDs. These implementations were realized on DataBricks (Community Edition, 2 cores) and Google Clouds (using 2 nodes).

By analyzing the results obtained after these various experiments, we can provide an answer to the problem we asked ourselves: it seems that the fastest, most efficient way of finding connected components within a graph on large scale data is to use RDDs when implementing the pseudo-code given in the reference paper on Google Cloud. Using Python or Scala seemed to matter very little in our results, and as such we also might recommend using Python for readability reasons.

Had we had more time and computation power, we would have tried to implement the pseudo code using DataFrames in Scala, but we decided not to prioritize this task given the fact that our results using DataFrames in Python were not very convincing. We would also

have liked to add more nodes on Google Cloud, and try to challenge the comparison when using four nodes or eight nodes for an even larger graph to test scalability for example.

1. Appendix

1.1. Codes

1.1.1. Python

RDDs

```
def iterate(a):
   valueList=[]
   m = a[0]
   for value in a[1]:
       if value<m:
           m = value
       valueList+=[value]
   if m<a[0]:
       yield((a[0], m))
       for val in valueList:
           if m!=val:
               incr.add(1)
               yield((val, m))

def iterate_CCF(graph):
   while incr.value>0 :
       incr.value=0
      #CCF_ITERATE
      #MAP
       graph = graph.flatMap(lambda x : [(x[0], x[1]), (x[1], x[0])])
       graph = graph.groupByKey()
       graph = graph.map(lambda x : (x[0], list(x[1])))
      #REDUCE
       graph = graph.flatMap(iterate)
      #CCF_DEDUP
       graph = graph.map(lambda x : ((x[0], x[1]), 0))
       graph = graph.reduceByKey(lambda x,y : 0)
       graph = graph.map(lambda x: x[0])
   return graph

def iterate_secondary_sorting(a):
   valueList=[]
   m = a[0]
   min_ = a[1][0]
   if min_ < m:
      yield (m, min_)
      for value in a[1]:
           incr.add(1)
           yield((value, min_))

def iterate_CCF_secondary_sorting(graph):
   global incr
   while incr.value>0 :
       incr.value=0
      #CCF_ITERATE
      #MA
       graph = graph.flatMap(lambda x : [(x[0], x[1]), (x[1], x[0])])
       graph = graph.groupByKey()
       graph = graph.map(lambda x : (x[0], list(x[1])))
      #REDUCE
```

```
        graph = graph.flatMap(iterate_secondary_sorting)
       #CCF_DEDUP
        graph = graph.map(lambda x : ((x[0], x[1]), 0))
        graph = graph.reduceByKey(lambda x,y : 0)
        graph = graph.map(lambda x: x[0])
    return graph
```

Dataframes

```
def iterate_map_df(df):
    newRow = df.select("val", "key")
    df1 = df.union(newRow)
    return df1

def iterate_reduce_df(df1):
    window = Window.orderBy("key","val").partitionBy("key")
    df_min = df1.withColumn("min", min("val").over(window))
    new_column_1 = expr( """IF(min > key, Null, IF(min = val, key, val))""")
    new_column_2 = expr("""IF(min > key, Null, min)""")
    new_df = (df_min.withColumn("new_key", new_column_1) \
                    .withColumn("new_val",new_column_2)) \
                    .na.drop() \
                    .select(col("new_key").alias("key"),col("new_val").alias(
             "val")) \
                    .sort("val", "key")
    df2 = new_df.distinct()
    return df2, df_min

def algo(df):
    counter = 1
    iteration = 0
    while counter!=0:
        iteration +=1
        df1 = iterate_map_df(df)
        df1.cache()
        df.unpersist()
        df, df_counter = iterate_reduce_df(df1)
        df.cache()
        df1.unpersist()
        df_counter = df_counter.withColumn("counter_col", expr("""IF(min >
key, 0, IF(min = val, 0, 1))""""))
        counter = df_counter.select(sum("counter_col")).collect()[0][0]
    return(df)
```

1.1.2. Scala (RDDs)

```
class count_process {
   var new_pairs = 0
   def iterate_1(rdd:RDD[(Int, Int)]):RDD[(Int, Int)] = {
     val rdd_1 = rdd.flatMap(x => List((x._1, x._2), (x._2,
x._1))).groupByKey().map(x => (x._1, x._2, x._2.min)).filter(x => x._1 >
x._3).map(x=> (x._1, x._2.filter(_ != x._3), x._3))
     new_pairs = rdd_1.map(_._2.size).sum().toInt

     return rdd_1.flatMap(x => List((x._1, x._3)) ++ x._2.map(y => (y,
x._3))).distinct()
   }

   def iterate_2(rdd:RDD[(Int, Int)]):RDD[(Int, Int)] = {
     val rdd_2 = rdd.flatMap(x => List((x._1, x._2), (x._2,
x._1))).groupByKey().mapValues(_.toList.sorted).filter(x => x._1 > x._2(0)).map(x
=> (x._1, x._2.slice(1, x._2.size), x._2(0)))

     new_pairs = rdd_2.map(_._2.size).sum().toInt

     return rdd_2.flatMap(x => List((x._1, x._3)) ++ x._2.map(y => (y,
x._3))).distinct()

   }

   def process(rdd:RDD[(Int, Int)], secondary_sorting: Boolean): RDD[(Int, Int)]
= {
     var i = 0
     var rdd_3 = rdd
     do {
       println(s"iteration $i")
       if (secondary_sorting) {
         rdd_3 = this.iterate_2(rdd_3)
       } else {
         rdd_3 = this.iterate_1(rdd_3)
       }
       i+=1
     }
     while(new_pairs > 0)

     return rdd_3
   }
 }
```