Database Management
Finding Connected Components in a Graph

CHRAIBI Ghali, DELSART Matthieu, VALENCONY Tim - 10/04/2024
Data Science for Business, École Polytechnique - HEC

Table of contents

Introduction

The paper *CCF: Fast and Scalable Connected Component Computation in MapReduce* [1], published in 2014, presented an efficient and scalable approach to finding connected components in graphs using the MapReduce framework. The objective of our project was to implement this CCF algorithm (Connected Component Finder) in 4 different programming frameworks: using Scala Spark and PySpark, and using RDDs (Resilient Distributed Datasets) and Spark DataFrames as main data structures. This algorithm was originally published in a research article in 2014 [1] and aims to identify the connected components within a graph. Our approach has combined these different frameworks with large-size datasets of different natures, enabling us to identify the advantages and weaknesses of the algorithm.

I. Description of the adopted Solution

As presented in the original paper, the CCF algorithm aims to find connected components in a graph using the MapReduce framework. It consists of two main MapReduce jobs: CCF-Iterate and

CCF-Dedup. The input to the algorithm is the list of all edges in the graph, and the output is a mapping from each node to its corresponding component ID.

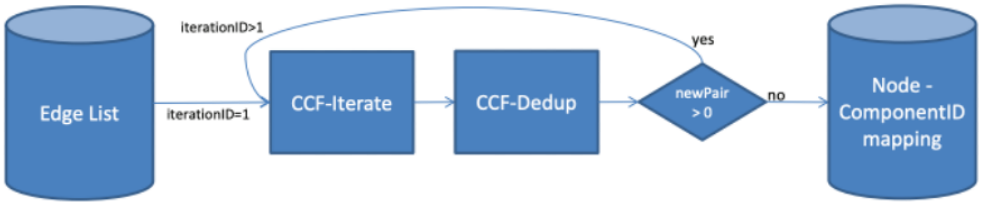

Figure 1: Connected Component Finder (CCF) Module
(from the original paper)

To explain clearly this algorithm, we will use as an example the graph below, which has two connected components.

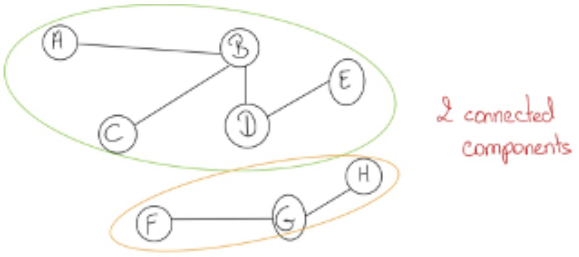

Figure 2: Example of graph

A. CCF-Iterate Job

This job generates adjacency lists for each node in the graph. It iterates over each node's adjacency list and emits pairs of (node, minID) where minID is the smallest node ID in the adjacency list. If there are multiple nodes in the adjacency list, additional pairs are emitted to ensure proper merging of connected components. The algorithm iterates until all nodes converge to their corresponding component IDs.

```
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)
```

Figure 3: CCF Iterate

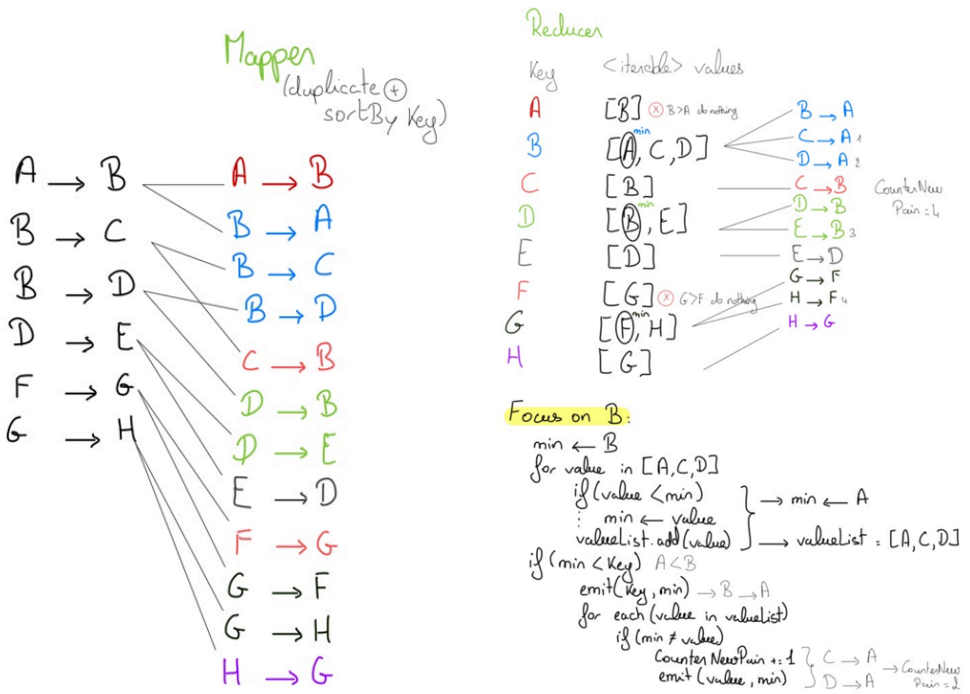

Figure 4: MapReduce detail in the CCF Iterate on our example

Map

For each edge in the input graph, the mapper emits two key-value pairs: **<node, neighbor>** and **<neighbor, node>**. This ensures that each node's adjacency list includes all its neighboring nodes.

Reduce

All values associated with the same key (node) are grouped together. For each node, the reducer iterates over its adjacent nodes and finds the minimum node ID (minValue) among them. If the minValue is smaller than the node's own ID, the reducer emits a pair <node, minValue>. This indicates that the node should merge with the component represented by minValue.

Additionally, for each adjacent node (except minValue), the reducer emits a pair **<adjacentNode, minValue>.** This ensures that all nodes in the same connected component have the same component ID. The reducer also increments a global counter (NewPair) by 1 for each emitted pair, indicating the discovery of new connections within components.

If the minValue is larger than the node's own ID, indicating that the node is already part of a smaller component, the reducer emits nothing for that node. After processing all nodes, the reducer checks the value of the NewPair counter. If it is zero, it indicates that no new connections were discovered, and the algorithm can terminate. Otherwise, it indicates that additional iterations are required to ensure convergence.

B. CCF-Dedup Job

CCF-Dedup

map(key, value)
  $temp.entity1 \leftarrow key$
  $temp.entity2 \leftarrow value$
  emit$(temp, null)$

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

Figure 5: CCF Dedup

After the CCF-Iterate phase, where pairs representing node-to-component mappings are emitted, duplicates may exist due to the iterative nature of the algorithm. The CCF-Dedup job aims to remove these duplicates to enhance the efficiency of the algorithm. In this job, the input consists of pairs representing node-to-component mappings generated by the CCF-Iterate phase. The mapper emits each pair with a dummy value. In the reduce phase, duplicates are automatically eliminated because reducers process keys in sorted order and receive all values associated with a key in a single group. Therefore, only unique pairs are emitted as the output of this phase, reducing unnecessary processing and I/O overhead. This could be easily translated to a **"rdd.distinct()"** on the iterated RDD.

C. Conclusion on the Number of Components

The algorithm iterates until no new connections are discovered during the CCF-Iterate phase. This convergence indicates that all nodes have converged to their final component IDs, and no further iterations are necessary. The algorithm terminates when the **NewPair** counter remains **zero** after processing all nodes in a single iteration. At this point, all connected components have been identified, and the algorithm outputs the final mapping from nodes to their corresponding component IDs. For the exemple provided, the algorithm converges in 4 iterations after CounterNewPair = 0 and results in the below schema having two distinct component IDs as expected.

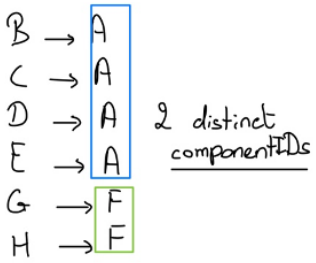

Figure 6: Final number of components in our example

II. Designed algorithms

A. Data import, cleaning and reading

The code snippets follow the approach explained before and implements the CCF approach on the web-Google[1] network from Stanford, having 875,713 nodes and 5,105,039 edges. The following code defines two functions for loading and cleaning network data stored in a CSV file for RDDs and Dataframes.

After defining these functions, the code then loads a CSV file containing graph data using the *load_table_rdd/df()* function and then cleans the loaded RDD/DF using the *clean_table_rdd/df()* function. The resulting RDD/DF is ready for further processing, such as applying the CCF algorithm.

- **RDDs**

```
def load_table_rdd(path):
    """
    Load data from a CSV file into an RDD.
    Assumes the CSV has a header which we skip.
    """
    rdd = sc.textFile(path)
    header = rdd.first() # extract header
    rdd = rdd.filter(lambda row: row != header) # filter out header
    return rdd

rdd = load_table_rdd('/FileStore/tables/web_Google.txt')

def clean_table_rdd(rdd):
    """
    Clean the RDD by splitting each line and converting to integer keys and
values.
    """
    return rdd.map(lambda line: line.split('\t')) \
              .map(lambda kv: (int(kv[0]), int(kv[1])))

rdd = clean_table_rdd(rdd)
```

1. **load_table_rdd(path)**: This function loads data from a CSV file into an RDD (Resilient Distributed Dataset). It assumes that the CSV file has a header, which it skips. Here's the breakdown of what each part of the function does:

    - sc.textFile(path): Loads the contents of the file located at the given path into an RDD of strings, where each string represents a line in the file.
    - header = rdd.first(): Extracts the header of the CSV file by getting the first line of the RDD.

[1] https://snap.stanford.edu/data/web-Google.html

- rdd.filter(lambda row: row != header): Filters out the header from the RDD by removing any lines that match the header string.
- Finally, the function returns the filtered RDD.

2. **clean_table_rdd(rdd)**: This function cleans the RDD by splitting each line into key-value pairs and converting them into integers. Here's the breakdown:

    - *rdd.map(lambda line: line.split('\t')):* Splits each line of the RDD by the tab character ('\t') to separate key-value pairs.
    - *.map(lambda kv: (int(kv[0]), int(kv[1]))):* Maps each key-value pair to a tuple where both the key and value are converted to integers using the int() function.
    - The function returns the cleaned RDD.

- **Dataframe**

```
def load_table_df(path):
   """
   return df from csv file
   """
   return spark.read.format("csv").option("header","true").option("inferSchema",
"true").load(path)

df = load_table_df('/FileStore/tables/web_Google.txt')

def clean_table_df(df):
   """
   clean the df imported
   """
   col = df.columns[0]
   return df.withColumn('key', split(df[col],
'\t').getItem(0).cast(IntegerType())) \
            .withColumn('value', split(df[col],
'\t').getItem(1).cast(IntegerType())) \
            .drop(col)

df = clean_table_df(df)
```

1. **load_table_df(path)**: This function returns a DataFrame (DF) loaded from a CSV file. It utilizes SparkSession's read method to read the CSV file, specifying options to treat the first row as a header and to infer the schema automatically.

2. **clean_table_df(df)**: This function cleans the DataFrame by splitting the values in the first column (presumably containing key-value pairs) and converting them to integers. Here's the breakdown:
    - It extracts the name of the first column.
    - It splits the values in the first column by the tab character ('\t') and extracts the first and second parts.

- It casts these extracted parts to integers and renames the columns as 'key' and 'value' and drops the original column.

B. CCF-Iterate Job

The following method encapsulates the pseudo code presented in Fig 3 and the main part of the iterative process where each iteration updates the component IDs of nodes based on their adjacency lists.

- **RDDs**

```
def fccg_iterate(rdd):
   # MAP
   rdd_mapped = rdd.union(rdd.map(lambda kv: (kv[1], kv[0])))

   # REDUCE
   # Create adjacency list and find minimum value in each list
   rdd_reduced = rdd_mapped.groupByKey().mapValues(lambda values:
(list(set(values)), min(values)))

   # Filter the rows that have min_value lower than key
   rdd_filtered = rdd_reduced.filter(lambda kv: kv[0] > kv[1][1])

   # Calculate the new pairs created
   new_count = rdd_filtered.map(lambda kv: len(kv[1][0]) - 1).sum() # -
rdd_filtered.count()

   # Prepare for next iteration
   rdd_final = rdd_filtered.flatMap(lambda kv: [(k, kv[1][1]) for k in kv[1][0]
+ [kv[0]] ]).distinct()

   return rdd_final, new_count
```

1. Mapping: **rdd_mapped = rdd.union(rdd.map(lambda kv: (kv[1], kv[0])))**

   In this step, the function first performs a mapping operation by swapping the key-value pairs. This operation creates a new RDD by concatenating the original RDD with a new RDD obtained by swapping the key-value pairs. This step ensures that each node's adjacency list includes all its neighboring nodes by adding both (node, neighbor) and (neighbor, node) pairs.

2. Reducing: **rdd_reduced = rdd_mapped.groupByKey().mapValues(lambda values: (list(set(values)), min(values)))**

   The function then performs a reducing operation by grouping the key-value pairs by key (node) and finding the minimum value (neighbor) in each group. It creates an adjacency list for each node and calculates the minimum value in each list.

3. Filtering: **rdd_filtered = rdd_reduced.filter(lambda kv: kv[0] > kv[1][1])**

This step filters out the pairs where the minimum value (neighbor) is smaller than the key (node), indicating that the node should merge with the component represented by the minimum value.

4. Calculating new pairs: **new_count = rdd_filtered.map(lambda kv: len(kv[1][0]) - 1).sum()**

   Here, the function calculates the number of new pairs created in the current iteration. It computes this by subtracting 1 from the length of each adjacency list (except the minimum value) and then summing them up across all nodes.

5. Preparing for the next iteration: **rdd_final = rdd_filtered.flatMap(lambda kv: [(k, kv[1][1]) for k in kv[1][0] + [kv[0]]]).distinct()**

   Finally, the function prepares the RDD for the next iteration by creating pairs where each node is associated with the minimum value in its adjacency list. It flattens the adjacency lists, adds the key node to each list, and removes duplicates to reproduce the CCF-Dedup job. It returns the resulting RDD (*rdd_final*) and the count of new pairs created (*new_count*), which are used to determine convergence in the CCF algorithm.

- **Dataframe**

```
 def fccg_iterate(df):
   # MAP
   map_df = df.union(df.select(col("value").alias("key"),
col("key").alias("value")))

   # REDUCE
   # Group by 'key' and aggregate values into a list, and find minimum value in
each group
   df_agg = map_df.groupBy("key").agg(
       collect_set("value").alias("adj_list"),
       array_min(collect_set("value")).alias("min_value")
       )

   # Filter the rows that have min_value lower than key
   df_agg_filtered = df_agg.filter(col("key")>col("min_value"))

   # Calculate the new pairs created (sum of values in the adjacency list)
   new_count = df_agg_filtered.select(_sum(size("adj_list"))).first()[0] -
df_agg_filtered.count()

   # Concat the key column with the adj_list one into one array
   df_concat = df_agg_filtered.select(concat(array(col("key")),
col("adj_list")).alias("adj_list"), col("min_value"))
   # Explode the 'values' list to separate rows and include 'min_value' for
comparison
   df_exploded = df_concat.select(
       explode("adj_list").alias("key"),
```

```
        col("min_value").alias("value")
    )

    # DEDUP
    df_final = df_exploded.distinct()

    return df_final, new_count
```

1. Mapping: **map_df = df.union(df.select(col("value").alias("key"), col("key").alias("value")))**

   In this step, the function performs a mapping operation similar to the RDD implementation by swapping the key-value pairs. It creates a new DataFrame by concatenating the original DataFrame with a new DataFrame obtained by swapping the 'key' and 'value' columns. This operation ensures that each node's adjacency list includes all its neighboring nodes by adding both (node, neighbor) and (neighbor, node) pairs.

2. Reducing: **df_agg = map_df.groupBy("key").agg(collect_set("value").alias("adj_list"), array_min(collect_set("value")).alias("min_value"))**

   The function then performs a reducing operation by grouping the DataFrame by the 'key' column and aggregating values into a list. It also finds the minimum value in each group, representing the smallest node ID in the adjacency list.

3. Filtering: **df_agg_filtered = df_agg.filter(col("key")>col("min_value"))**

   This step filters out the rows where the minimum value (neighbor) is smaller than the key (node), indicating that the node should merge with the component represented by the minimum value.

4. Calculating new pairs: **new_count = df_agg_filtered.select(_sum(size("adj_list"))).first()[0] - df_agg_filtered.count()**

   Here, the function calculates the number of new pairs created in the current iteration. It computes this by summing the sizes of all adjacency lists and subtracting the count of filtered rows.

5. Preparing for the next iteration: **df_final = df_exploded.distinct()**

   Finally, the function prepares the DataFrame for the next iteration by removing duplicate pairs using the distinct() function. The function returns the resulting DataFrame (df_final) and the count of new pairs created (new_count), which are used to determine convergence in the CCF algorithm.

C. All-in-all workflow

These code snippets represent the main computation loop for finding connected components using the CCF algorithm. It takes an RDD/DF as input and iteratively applies the CCF-Iterate job until no new pairs are discovered, indicating that all nodes have converged to their final component IDs.

- **RDDs**

```
def compute_fccg(rdd):
    """Main computation loop to find connected components."""
    nb_iteration = 0

    while True:
        nb_iteration += 1

        rdd, count = fccg_iterate(rdd)

        print(f"Number of new pairs for iteration #{nb_iteration}:\t{count}")
        if count == 0:
            print("\nNo new pair, end of computation")
            break

    return rdd

final_rdd = compute_fccg(rdd)

final_rdd.map(lambda x: x[1]).distinct().count()
```

- **Dataframes**

```
def compute_fccg(df):
    """Main computation loop to find connected components."""
    nb_iteration = 0

    while True:
        nb_iteration += 1

        df, count = fccg_iterate(df)

        print(f"Number of new pairs for iteration #{nb_iteration}:\t{count}")
        if count == 0:
            print("\nNo new pair, end of computation")
            break

    return df

final_df = compute_fccg(df)

final_df.select('value').distinct().count()
```

The mains components of the **compute_fccg()** method are the following:

- **while True:** The infinite loop that continues until the termination condition is met (i.e., no new pairs are discovered during an iteration).
- **rdd, count = fccg_iterate():** Applies the CCF-Iterate job previously described to the input, returning the updated RDD/DF and the number of new pairs discovered during the iteration.
- **if count == 0:** Checks if the number of new pairs discovered during the current iteration is zero, indicating that no new connections were found and the algorithm can terminate.
- **Break:** Terminates the while loop when no new pairs are discovered during an iteration.
- **return rdd**: Returns the final RDD/DF containing the node-to-component mappings after the computation has ended.
- **final_rdd/df.distinct().count():** Counts the number of distinct connected components in the final RDD by extracting the component IDs from the node-to-component pairs, removing duplicates, and counting the remaining unique component IDs.

III. Experimental Analysis

Apart from small, hand-designed datasets, our experimental analysis focused on 4 main datasets, all based on web graphs:

- Notre Dame web graph [2]
- Google web graph [3]
- Stanford web graph [4]
- BerkStan web graph [5]

Their main characteristics are summarized below:

Dataset	**Notre Dame**	**Google**	**Stanford**	**BerkStan**
Nodes	325 729	875 713	281 903	685 230
Edges	1 497 134	5 105 039	2 312 497	7 600 595
Size	21Mo	74Mo	32Mo	101Mo

Figure 7: Main characteristics of the experimental graphs

Our experimental analysis occurred in two main stages:

1. **Comparing the 4 different implementations and their performances.** For this stage, we only used the first two datasets (Notre Dame and Google) as we were limited by Databricks on the other two ones (see below)

2. **Comparing the performances of the same implementation on the 4 different models.** Here, the objective was to better understand in the graphs what makes the algorithm fast to operate or not. We chose to use Scala on RDDs as it is the fastest of the four implementations.

A. Comparing the 4 different implementations

Due to too high execution times for the last two datasets, we focused on the first two datasets. The execution times in each framework are given below.

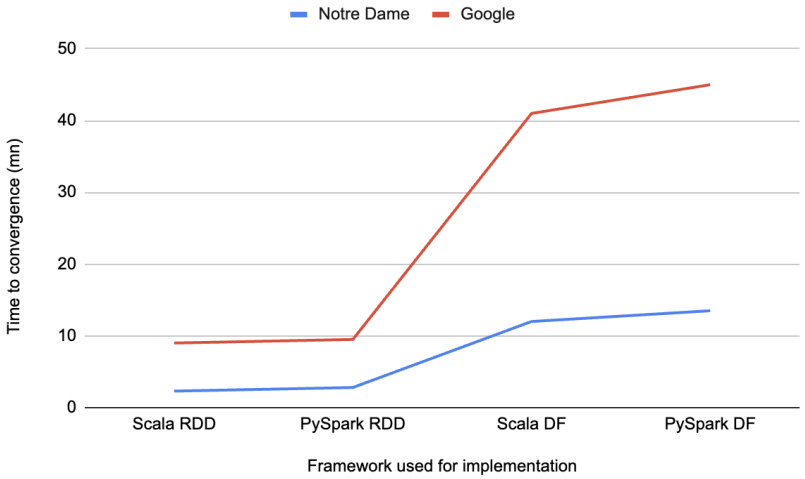

Figure 8: Execution time of the 4 algorithms on the Notre Dame and Google web graphs datasets

From these tests, three main observations stand out:

- **The time execution ratio between the two datasets is stable across frameworks**, with Google being approximately 4x longer to converge than Notre Dame.
- On the same data structure, **PySpark is slightly longer to run than Scala** (about 10%). This can be explained by the fact that Java is the native language of Apache Spark, entailing slight advantages such as static typing (allowing the compiler to optimize code better) and JVM overhead (Scala directly compiles to Java Virtual Machine bytecode whereas PySpark has to use an additional layer).
- On our algorithms, with the same language used, **RDDs are approximately 5x longer to run than DataFrames**. This can be explained by the fact that the algorithm tasks are relatively simple and that no uses of DataFrames, such as native column names, are used in it.

B. Comparing the 4 datasets

Here, we used the fastest framework, Scala on RDDs, to compare execution time across the four datasets. The graph below shows the different times and number of iterations of the four datasets. As we were using Databricks, where the clusters time out after 1h, we were not able to go until the end of the execution for the Stanford dataset.

Dataset	**Notre Dame**	**Google**	**Stanford**	**BerkStan**
Execution time	2,5mn	9mn	> 1h	47mn
Iterations to convergence	7	8	> 14	13

Figure 9: Number of iterations and execution time of Scala RDD on the 4 datasets

Quite surprisingly, the execution time and number of iterations needed to converge does not seem correlated to the number of nodes, edges, or the size of the dataset, since the Stanford one is the second lightest one in all these aspects and still takes much longer than the others to run. To better understand this, we decided to look at the execution times of each iteration. For readability, we are displaying a graph for each dataset.

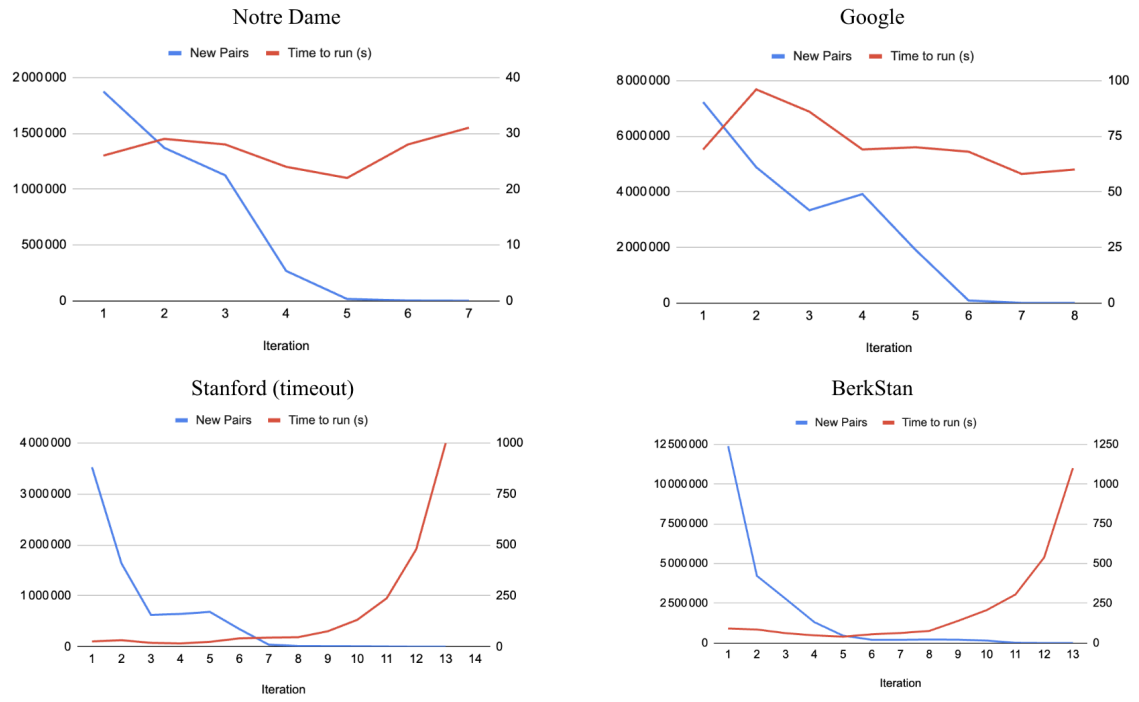

Figure 9: Number of new pairs and execution time for each iteration on the 4 datasets, in Scala RDD
Note that the left y-axis concerns the number of new pairs and the right y-axis concerns the execution time

In all the datasets, the number of new pairs at each iteration is overall decreasing exponentially, with some plateaux sometimes appearing. However, we can see that the execution time varies a lot between the different datasets. Before the ~8th iteration, the execution time per iteration is globally steady and roughly proportional to the size of the dataset (~28s for Notre Dame vs ~65s for Google for instance. However, it then increases exponentially for the two datasets that did not converge after that stage, Stanford and BerkStan, and at a stronger rate for Stanford (x2 at each iteration) than for Berkstan (x1.5 each iteration). From our experiments, we cannot say if this exponential increase after the 8th iteration is due to the number of iterations (and would have occurred to the other datasets if the algorithms had taken longer to converge for them), or if it is due to the inner structure of the last two datasets. However, it is interesting to note that, after the 8th iteration, the number of stages in each Spark job is regularly multiplied by 2 in the last 2 datasets: it is for example $32,768 = 2^{15}$ at the 13th iteration in the Stanford dataset.

We thus see that the algorithm execution time does not seem strongly linked to the number of nodes, edges, and size of the datasets. However, its execution time sometimes increases exponentially, although we are unable to determine if this is due to some properties of the datasets it is being applied to (inner structure complexity, for instance), or if this exponential increase always occurs after the 8th iteration.

IV. Comments about the implementation and Conclusion

Although we have been limited in our experimentations by Databricks rules on execution times, we have been able to implement the CCF algorithm in the 4 different frameworks and test on 4 different datasets. This has enabled us to assess the comparative performance of Scala and PySpark, whether it was using RDDs or DataFrames.
We would have liked to do more tests, more specifically using Google Cloud. However, our credits expired (we were not able to redeem them anymore) and we were thus unable to run anything on it.

Overall we see that, if the number of workers is limited, the number of jobs per iteration increases exponentially, and this is a real problem for datasets that need bigger numbers of iterations to converge. Therefore, finding a way to reduce that number using an improved CCF algorithm would be optimal, even if it means it is less performant for small datasets. This is supported by the fact that the number of iterations for CC-MR [6] compared in the paper needs less iterations to converge, and thus takes less time than CCF for larger datasets.

All in all, in this report we have been able to correctly implement the *Connected Component Finder* defined in the paper at the beginning of the report. The algorithms provided are easily reproducible in another environment, and one can choose from four different methods depending on the needs. Theoretically, the Map Reduce approach in the paper sets the stage for scalability to billions of nodes and edges, provided we have enough Drivers and Workers to support our implementation. This is a significant improvement to other more naive algorithms introduced previously in the literature.

The full code of our implementation can be found at this Github Repository.

V. APPENDIX

A. References

1. Kardes, H., Agrawal, S., Wang, X., & Sun, A. (2014). CCF: Fast and scalable connected component computation in MapReduce. IEEE Transactions on Knowledge and Data Engineering, 26(10), 2455-2467.

2. Notre Dame web graph: R. Albert, H. Jeong, A.-L. Barabasi. Diameter of the World-Wide Web. Nature, 1999.

3. Google web graph: J. Leskovec, K. Lang, A. Dasgupta, M. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics 6(1) 29--123, 2009. Retrieved at: https://snap.stanford.edu/data/web-Google.html

4. Stanford web graph: J. Leskovec, K. Lang, A. Dasgupta, M. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics 6(1) 29--123, 2009. Retrieved at: https://snap.stanford.edu/data/web-Stanford.html

5. BerkStan web graph: J. Leskovec, K. Lang, A. Dasgupta, M. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics 6(1) 29--123, 2009. Retrieved at: https://snap.stanford.edu/data/web-BerkStan.html

6. T. Seidl, B. Boden, and S. Fries, "Cc-mr finding connected components in huge graphs with mapreduce," in Machine Learning and Knowledge Discovery in Databases, ser. Lecture Notes in Computer Science, P. Flach, T. Bie, and N. Cristianini, Eds. Springer Berlin Heidelberg, 2012, vol. 7523, pp. 458–473. Available: http://dx.doi.org/10.1007/978-3-642-33460-3_35

B. Code - Scala RDD

Unset

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD

val conf = new SparkConf().setAppName("FCCG RDD Implementation")
val sc = SparkContext.getOrCreate(conf)

def loadTableRDD(path: String): RDD[String] = {
val rdd = sc.textFile(path)
```

```
val header = rdd.first() // Extract header
rdd.filter(row => row != header) // Filter out the header
}

var rdd = loadTableRDD("/FileStore/tables/web_NotreDame.txt")

def cleanTableRDD(rdd: RDD[String]): RDD[(Int, Int)] = {
rdd.map(_.split("\t")).map(kv => (kv(0).toInt, kv(1).toInt))
}

var clean_rdd = cleanTableRDD(rdd)

def fccgIterate(rdd: RDD[(Int, Int)]): (RDD[(Int, Int)], Double) = {
// MAP
val rddMapped = rdd.union(rdd.map(kv => (kv._2, kv._1)))

// REDUCE
// Create adjacency list and find minimum value in each list
val rddReduced = rddMapped.groupByKey().mapValues(values => {
val valuesSet = values.toSet
(valuesSet, valuesSet.min)
})

// Filter the rows that have min_value lower than key
val rddFiltered = rddReduced.filter(kv => kv._1 > kv._2._2)

// Calculate the new pairs created
val newCount = rddFiltered.map(kv => kv._2._1.size - 1).sum()

// Prepare for next iteration
val rddFinal = rddFiltered.flatMap(kv => (kv._2._1 + kv._1).map(k =>
(k, kv._2._2))).distinct()

(rddFinal, newCount)
}
```

```
def computeFccg(rdd: RDD[(Int, Int)]): RDD[(Int, Int)] = {
var nbIteration = 0
var loop = true
var new_rdd = rdd

while (loop) {
nbIteration += 1

val result = fccgIterate(new_rdd)
val count = result._2
new_rdd = result._1

println(s"Number of new pairs for iteration #$nbIteration:\t$count")
if (count == 0) {
println("\nNo new pair, end of computation")
loop = false
}
}
new_rdd
}
var final_rdd = computeFccg(clean_rdd)

// Get total number of connected components
final_rdd.map(_._2).distinct().count()
```

C. Code - PySpark RDD

Python

```
from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession
```

```
def load_table_rdd(path):
"""
Load data from a CSV file into an RDD.
Assumes the CSV has a header which we skip.
"""
rdd = sc.textFile(path)
header = rdd.first() # extract header
rdd = rdd.filter(lambda row: row != header) # filter out header
return rdd


conf = SparkConf().setAppName("FCCG RDD Implementation")
sc = SparkContext.getOrCreate(conf=conf)

# Load data into an RDD
rdd = load_table_rdd('/FileStore/tables/web_Google.txt')

def clean_table_rdd(rdd):
"""
Clean the RDD by splitting each line and converting to integer keys
and values.
"""
return rdd.map(lambda line: line.split('\t')) \
.map(lambda kv: (int(kv[0]), int(kv[1])))

rdd = clean_table_rdd(rdd)

def fccg_iterate(rdd):
# MAP
rdd_mapped = rdd.union(rdd.map(lambda kv: (kv[1], kv[0])))
# REDUCE
# Create adjacency list and find minimum value in each list
rdd_reduced = rdd_mapped.groupByKey().mapValues(lambda values:
(list(set(values)), min(values)))
# Filter the rows that have min_value lower than key
rdd_filtered = rdd_reduced.filter(lambda kv: kv[0] > kv[1][1])
```

```
# Calculate the new pairs created
new_count = rdd_filtered.map(lambda kv: len(kv[1][0]) - 1).sum() # -
rdd_filtered.count()
# Prepare for next iteration
rdd_final = rdd_filtered.flatMap(lambda kv: [(k, kv[1][1]) for k in
kv[1][0] + [kv[0]] ]).distinct()
return rdd_final, new_count
```

```
def compute_fccg(rdd):
"""Main computation loop to find connected components."""
nb_iteration = 0
while True:
      nb_iteration += 1
      rdd, count = fccg_iterate(rdd)

      print(f"Number of new pairs for iteration
      #{nb_iteration}:\t{count}")
      if count == 0:
            print("\nNo new pair, end of computation")
            break
return rdd
```

```
final_rdd = compute_fccg(rdd)
```

```
# Number of connected components in the graph
final_rdd.map(lambda x: x[1]).distinct().count()
```

D. Code - Scala DataFrame

Unset

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._
```

```
import org.apache.spark.sql.types.IntegerType
import org.apache.spark.sql.DataFrame

def loadTableDF(spark: SparkSession, path: String) = {
spark.read.format("csv")
.option("header", "true")
.option("inferSchema", "true")
.load(path)
}

def cleanTableDF(df: org.apache.spark.sql.DataFrame) = {
val colName = df.columns(0) // Assuming there's only one column in
the original CSV
df.withColumn("key", split(col(colName),
"\t").getItem(0).cast(IntegerType))
.withColumn("value", split(col(colName),
"\t").getItem(1).cast(IntegerType))
.drop(colName)
}

val spark = SparkSession.builder()
.appName("Data Loader and Cleaner")
.getOrCreate()

// Assuming SparkSession is initialized as 'spark'
import spark.implicits._

// Load the DataFrame
val df = loadTableDF(spark, "/FileStore/tables/web_Google-4.txt")

// Clean the DataFrame
val cleanedDF = cleanTableDF(df)

// Show some rows to verify
```

```
cleanedDF.show()


def fccgIterate(df: DataFrame): (DataFrame, Long) = {
// MAP
val mapDF = df.union(df.select($"value".alias("key"),
$"key".alias("value")))

// REDUCE
// Use collect set for deduplication !
val dfAgg = mapDF.groupBy($"key").agg(
collect_set($"value").alias("adj_list"),
array_min(collect_set($"value")).alias("min_value")
)

val dfAggFiltered = dfAgg.filter($"key" > $"min_value")

val newCount =
dfAggFiltered.select(sum(size($"adj_list"))).first().getLong(0) -
dfAggFiltered.count()

// Concat the 'key' column with the 'adj_list' one into one array
val dfConcat = dfAggFiltered.select(concat(array(col("key")),
col("adj_list")).alias("adj_list"), col("min_value"))

// Explode the 'adj_list' array to separate rows and include
'min_value' for comparison
val dfExploded = dfConcat.select(
explode(col("adj_list")).alias("key"),
col("min_value").alias("value")
)

// Deduplicate the data
val dfFinal = dfExploded.distinct()
```

```
(dfFinal, newCount)
}
```

```
def computeFCCG(df: DataFrame): DataFrame = {
var nbIteration: Int = 0
var loopDF: DataFrame = df
var count: Long = 1

while (count != 0) {
nbIteration += 1
val resultTuple: (DataFrame, Long) = fccgIterate(loopDF)
loopDF = resultTuple._1
count = resultTuple._2

println(s"Number of new pairs for iteration #$nbIteration:\t$count")
}

println("\nNo new pair, end of computation")
return loopDF // Exit the loop and return the DataFrame
}
```

```
val final_df = computeFCCG(cleanedDF)

// Calculate Number of connexted components in graph

final_df.select($"value").distinct().count()
```

E. Code - PySpark DataFrame

Python

```
import time
```

```
import pyspark
from pyspark.sql import SparkSession
from pyspark import SparkConf
from pyspark.sql.functions import *
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import sum as _sum, size
```

```
def load_table_df(path):
"""
return df from csv file
"""
return
spark.read.format("csv").option("header","true").option("inferSchema"
, "true").load(path)
```

```
df = load_table_df('/FileStore/tables/web_Google.txt')
```

```
def clean_table_df(df):
"""
clean the df imported
"""
col = df.columns[0]
return df.withColumn('key', split(df[col],
'\t').getItem(0).cast(IntegerType())) \
.withColumn('value', split(df[col],
'\t').getItem(1).cast(IntegerType())) \
.drop(col)
```

```
df = clean_table_df(df)
```

```
def fccg_iterate(df):
# MAP
map_df = df.union(df.select(col("value").alias("key"),
col("key").alias("value")))
```

```
# REDUCE
# Group by 'key' and aggregate values into a list, and find minimum
value in each group
df_agg = map_df.groupBy("key").agg(
collect_set("value").alias("adj_list"),
array_min(collect_set("value")).alias("min_value")
)

# Filter the rows that have min_value lower than key
df_agg_filtered = df_agg.filter(col("key")>col("min_value"))

# Calculate the new pairs created (sum of values in the adjacency
list)
new_count = df_agg_filtered.select(_sum(size("adj_list"))).first()[0]
- df_agg_filtered.count()

# Concat the key column with the adj_list one into one array
df_concat = df_agg_filtered.select(concat(array(col("key")),
col("adj_list")).alias("adj_list"), col("min_value"))
# Explode the 'values' list to separate rows and include 'min_value'
for comparison
df_exploded = df_concat.select(
explode("adj_list").alias("key"),
col("min_value").alias("value")
)

# DEDUP
df_final = df_exploded.distinct()

return df_final, new_count
```

```
def compute_fccg(df):
```

```
"""Main computation loop to find connected components, removing
global variable usage."""
nb_iteration = 0
while True:
nb_iteration += 1
df, count = fccg_iterate(df)

print(f"Number of new pairs for iteration #{nb_iteration}:\t{count}")
if count == 0:
print("\nNo new pair, end of computation")
break
return df
```

```
final_df = compute_fccg(df)
```

```
# Number of clusters
final_df.select('value').distinct().count()
```