- Database Management System -
Connected Component Finder
Final Report

Hugo Ehlinger, Louis Bertolotti, Maxime Redstone Leclerc

March 2021

1 Introduction

Graphs are structures consisting of a set of vertices (or nodes) connected through edges. They are used in a wide variety of settings to model social networks, blockchain, web pages and even biological systems. Finding connected components in a graph enables to reorganize the graph in independent clusters.

The aim of this project is to design an algorithm able to find connected components within graphs that model a Big Data setting, that is to say on files representing graphs made of billions of nodes and edges. Parallelization using distributed systems provides an efficient framework to extract meaningful information when dealing with large graphs.

This final report explains the method followed to reach the objective and provides a performance analysis of the algorithms implemented. Section 2 describes the solution based on the work developed in [1]. Section 3 focuses on the various implementations, in Python and Scala using RDD and Dataframes of the algorithm. Finally, sections 4 and 5 evaluate the performances of our approach using various graph sizes.

This project is supported by a public Github repository that contains the entire code we will refer to in this report. This repository can be found at `https://github.com/MaximeRedstone/graphsMapReduce`.

Contents

2 Adopted Solution

The authors of [1] presented a fast and scalable connected component computation using MapReduce in their record linkage pipeline (cleaning, blocking, pair-wise linkage and clustering). The aim of their study was to regroup records belonging to a single entity found in different databases. We focus on the last step of their pipeline (clustering) which combines record pairs into connected components. Their methodology outperforms other algorithms such as Pegasus [2], CC-MR [3] which is why it is taken as a basis for this work. Indeed they demonstrated, using a 75Mb dataset that their algorithm was 10 times faster than the PEGASUS approach and performed nearly as well as the CC-MR approach. CCF took a couple more iterations to reach a solution than CC-MR but the authors argue that it is due to the initialisation of the map/reduce tasks. CCF was also successfully performed on a very large graph composed on 92B edges illustrating the scalability of the algorithm.

The pseudo-code is shown in Algorithms 1 and 2.

Algorithm 1: CCF-Iterate with secondary sorting

```
  Input: Connected edges (A B) in .txt format
1 map(key, value)
      emit(key, value)
      emit(value, key)
2 reduce(key, <iterable >values)
      minValue ← $values.next()$
      if (minValue <key) then
          emit(key, minValue)
          foreach value ∈ $values$ do
              Counter.NewPair.increment(1)
              emit(value, minValue)
          end
      end
```

Algorithm 2: CCF-Dedup

```
1 map(key, value)
      temp.entity1 ← key
      temp.entity2 ← value
      emit(temp, null)
2 reduce(key, <iterable >values)
      emit(key.entity1, key.entity2)
```

3 Implementation

We aim to provide four different implementations of the MapReduce algorithms described above. The authors of [2] originally deployed their algorithm on Hadoop using the HDFS file system to support the parallelization of the data and computations. The use of Hadoop raises the following main issues:

- After each map or reduce transformation, the output must be written on the disk using HDFS. This operation is necessary for mappers and reducers to communicate with each others. Writing on disk is a time costly operation.
- The MapReduce framework offers a very limited set of instructions that makes difficult the implementation of complex algorithms.

Considering both these limitations, our study aims at deploying the MapReduce algorithm described in Section 2 on Spark. Spark enables writing intermediary results on the RAM achieving a 10 to 100 times gain in performance. This results from the fact that writing on the RAM is extremely fast as opposed to writing on disk. Moreover, Spark provides two kinds of abstract data structures: Resilient Distributed Datasets (RDDs) and DataFrames (DFs) that make the code easier to write and read.

Implementations both in Scala and Python using both RDDs and DFs have been developed and are presented below. The entire code for each implementation is presented in the Appendix. The entire code for the project can be found at `https://github.com/MaximeRedstone/graphsMapReduce`.

3.1 Key Steps

A step-by-step description of the pseudo-code implemented is described below.

- The graph is read as a text file (.txt) in which each line represents an edge. The expected format is ”node1 node2”.
- The file is split on whitespaces and edges are stored as tuples (node1, node2)
- An accumulator is set to any non-0 value. Accumulators are global variables which aggregate a count across all workers used during the parallelization process.
- The following operations are triggered while the accumulator value is non-0:
  - The accumulator is reset to 0.
  - CCF Iterate:

        * Mapper: Each edge (node1, node2) is duplicated to appear also as (node2, node1). The outcome is then grouped by node so that each node is associated to the list of nodes that are directly connected to it. Output: (node, list(nodes connected))
        * Reducer: In the case when a node's id is bigger than any of its connections, all connections are rewritten as connections towards the smallest node's id. In this case, the accumulator is incremented by $(number\_of\_connections\_rewritten - 1)$. This enables, after enough iterations, to regroup all connections within a cluster towards the node with the smallest id.
    - CCF Dedup: This second part of the algorithm is only an efficient way of removing duplicates using the $groupByKey$ command of Spark.
        * Mapper: Both nodes of the edge are merged so that $groupByKey$ considers the whole edge as key
        * Reducer: The unique edges need to be unpacked as two separate nodes again.

The core of the algorithm is the CCF Iterate part. The strategy of the algorithm is to redirect connected components towards the node with the smallest id among the group of nodes to identify clusters of nodes. This way, the output of the algorithm is also expressed as edges of the form (node1, node2). *Node2* is the node with the smallest id within the cluster. The corresponding *node1s* are all the nodes belonging to the same cluster, that is to say the nodes being directly or indirectly connected to *node2*. Clusters are strictly distinct so that there is no connection between clusters.

Figure 1 shows a toy graph provided by the authors of [1] to illustrate their algorithm. The original connections are A-B, B-C, B-D, D-E, F-G, G-H. The algorithm regroups these connections as two clusters identified with the smallest ids, here A and F. Therefore, the output of the algorithm is B-A, C-A, D-A, E-A, G-F, H-F.

3.2 Main differences between the four implementations

Overall, both Scala implementations are translations of the PySpark implementations. Nevertheless, some structural differences between Scala and PySpark induce significant differences in the algorithm:

- On PySpark any operation, like $collect$ for instance, triggers the precedent transformations and overrides the content of the intermediary variables. Nevertheless, in the Dataframe implementation in Scala, collecting does not override the intermediary variables. That is why our implementation catches the return of the $collect$ operation directly into a Dataframe, which the PySpark implementation does not require such procedure. This may have an impact on the difference in performances between the two implementations as highlighted in Section 5.

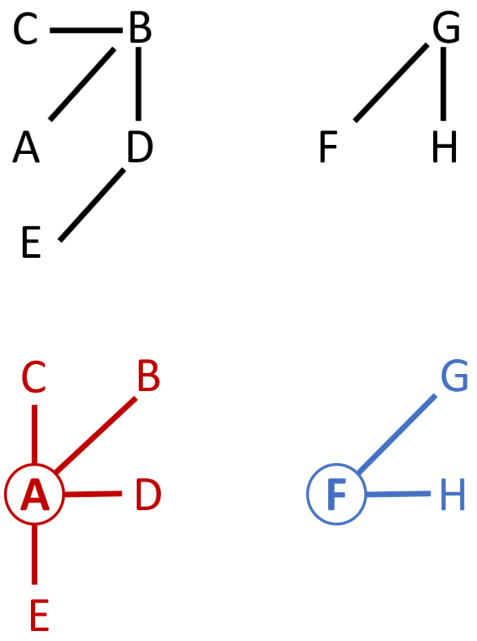


Figure 1: Representation of CCF-Iterate Algorithm for a toy graph.

- On the Dataframe implementation in Scala, we noticed that our user-defined-function ($it - reduce$) was called several times per record as documented in [4]. This issue did not occur in the PySpark equivalent. In order to avoid a performance loss due to useless extra calls to the user-defined-function, we decided to register the function as non deterministic, so that the function is executed only once (as Spark expects that running it several times could yield different results).

3.3 Improvements Exploration

While implementing the pseudo code shown in Algorithms 1 and 2 we explored different versions of the code as shown in Figure 2.

- The pseudo-code for the CCF-Iterate algorithm increments the accumulator 1 by 1 in a loop. Instead, we propose to increment the counter once by the length of the list to avoid useless operations. As described in [5], the Python built-in function $len(list)$ returns meta-data of the list and therefore does not require parsing in any way. It is then much more efficient to increment the counter only once (Subfigures 2 a) and c)).
- The authors of [1] recommend to use a version of CCF-Iterate (Algorithm 1)

in which values are sorted, so that extracting the minimum value reduces to extracting the first value. Nevertheless, we question the need to sort the full list of values when extracting the minimum is the only operation required. This is why a version in which we extract the minimum has been tested using Numpy's $argmin(x)$ built-in list capability (Subfigures 2 a) and b)). Indeed, efficient sorting algorithms manage to achieve a $\mathcal{O}(x * log(x))$ complexity while the $argmin(x)$ function has a $\mathcal{O}(x)$ complexity with x being the size of the list to sort.

- The original algorithm of CCF-Iterate uses a for-loop to emit the additional pairs after the $emit(key, min)$. We have tried to develop an alternative that avoids this for-loop by using the $zip(iterable1, iterable2)$ Python function (Subfigures 2 b) and c)).
- Dataframes implement the SQL's $dropDuplicates$ function that is likely to replace the Dedup Map and Reduce operations. Similarly RDDs implement the $.distinct()$ operation that is likely to be better optimized than the three operations of the Dedup algorithm.

Most of these features are easily implementable in Scala as well. The performance analysis in Section 5 provides the outcome of these tests.

4 Experimental Analysis methodology

In order to evaluate our approach we devised a pipeline to understand the scalability, strong and weak points of our implementations. Subsection 4.1 describes a tool we created to generate graphs of any given size. Subsection 4.2 presents our methodology to analyse Spark logs automatically.

4.1 Graphs Generator

The authors of [1] tested their algorithm on graph sizes ranging 5 to 200 MB. In order to provide a more accurate picture of the scalability of our implementation, we devised a graph generator. Developing our own graph generator answers the following issues:

- This generator takes as input the final size of the file. Therefore, it enables to run the algorithm on graphs which sizes increase linearly to perform an accurate analysis of the complexity of the algorithm.
- This generator enables the user to chose an approximate number of edges per node. In this way, this parameter is fixed and does not influence the performance analysis, while using graphs computed from various sources could induce biais.

```python
def reduce_ccf_sorted(x):
  key = x[0]
  values = x[1]
  min_value = values.pop(0)
  ret = []
  if min_value < key:
    ret.append((key, min_value))
    for value in values:
      accum.add(1)
      ret.append((value, min_value))
  return (ret)


it_groupby = it_map.groupByKey().mapValues(lambda x: sorted(x))
it_reduce = it_groupby.flatMap(lambda x: reduce_ccf_sorted(x))
```

(a) Sort values and take first element.

```python
def reduce_ccf_min(x):
  key = x[0]
  values = x[1]
  min_value = values.pop(values.index(min(values)))
  ret = []
  if min_value < key:
    ret.append((key, min_value))
    accum.add(len(values))
    for value in values:
      ret.append((value, min_value))
  return (ret)

it_groupby = it_map.groupByKey().mapValues(list)
it_reduce = it_groupby.flatMap(lambda x: reduce_ccf_min(x))
```

(b) Find index of minimal value.

```python
def reduce_ccf_zip(x):
  key = x[0]
  values = x[1]
  min_value = values.pop(values.index(min(values)))
  if min_value < key:
    size = len(values)
    accum.add(size)
    values.insert(0, key)
    min_value_list = [min_value] * (size + 1)
    return list(zip(values, min_value_list))
  return []

it_groupby = it_map.groupByKey().mapValues(list)
it_reduce = it_groupby.flatMap(lambda x: reduce_ccf_zip(x))
```

(c) Drop **foreach** loop using python's zip capability.

Figure 2: Improvements tested compared to base pseudo-code.

The generator is coded in C for faster performances using direct system calls and sound memory management. The generator can be found in the **generator** folder of the Github Repository and is run as follows.

```
make re
./generator 10M graph_10M.txt 5
```

Source Code 1: Running Generator Application

User can pass three graph parameters:

- Size: Int + Unit ([B]yte, [K]ilobyte, [M]egabyte, [G]igabyte are supported)
- Filename: default to graph_($size).txt.
- Average number of edges per node: usually an Int between 5 and 20.

Upon successful checks of user defined parameters, the program converts the given input size to the exact size in bytes. It then calculates the approximate number of nodes required to reach a file of the expected size while satisfying the user constraint on the number of edges per node. It then generates random edges of the format: $node1$ as int SPACE $node2$ as int.

The generator can then be launched and its pseudo-code is shown in Algorithm 3.

Algorithm 3: Graph Generator

```
  Pre-compilation setting: buffer_size = 256M
  Input: User Parameters e.g. 10M, filename, edges per node
1 Find the number of nodes necessary to provide a graph of the expected size
    with required edges per node
2 while file_size <user_value do
3     Write new edge (e.g. 1030 2040) to buffer
4     if current_buff_size >buffer_size then
5         Write buffer content to disk
6         Free buffer
      end
  end
  Output: Text File
```

Performances of our generator tool are shown in Table 1. Empirical evidence points out to a $\mathcal{O}(N)$ complexity. This insures we will be able to generate a large number of graph files to analyse our MapReduce implementations. A larger buffer would reduce the number of write system calls, thereby improving the time performance of the algorithm. Nevertheless, it would solicit more RAM.

Graph Size	Generating Time (seconds)
10M	1
100M	6
500M	32
1G	65
10G	739

Table 1: Time to generate graph text files.

4.2 Log Analyser

In order to help analyse the performances of our MapReduce implementations, we make use of spark logs. Databricks provides an ergonomic Spark UI interface but we decided to use directly the raw log files. This enables us to develop customized data visualisation features that will enhance the performance analysis. The raw logs of a Databricks cluster are stored in the cluster as a single nested json file. We developed a Python script to be launched directly in the cluster to parse this json file and filter the relevant information. The logs data is then stored as multiple csv files that replicate the structure of a SQL-like database as presented in Figure 3.

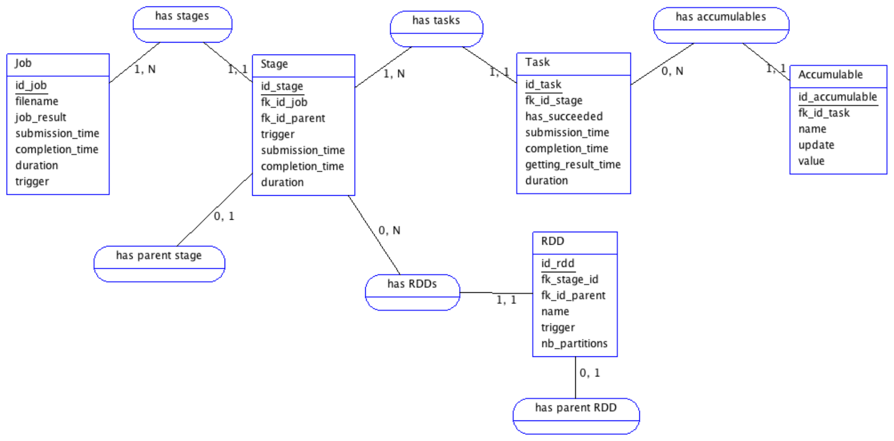


Figure 3: Merise Diagram of Databricks' Logs File.

Below is a short description of each table:

- log_jobs.csv: The jobs are the parents of all subsequent tasks in the cluster. A job is triggered at each collect action, as described in Subsection 3.1. Each job is associated to its original graph. Relevant information such as the duration of the job is collected.
- log_stages.csv: The stages are children of the jobs. Stages are triggered by the

GroupByKey transformation and the collect action. As for jobs, relevant information such as the duration of the stage is collected.

- log_tasks.csv: The tasks are children of the stages. Each triggers as many tasks as cores of the cluster. The analysis of the tasks logs will enable to better understand the parallelization of the algorithm.
- log_rdds.csv: RDDs are children of the stages. They show intermediary states of the data across the algorithm.
- log_accumulables.csv: The logs on the accumulable show the intermediary updates by the tasks and enables to measure the performance of the parallelization.

An additional template of logs has been designed to isolate the time performance of each step of the algorithm: CCF-Map, CCF-groupby, CCF-Reduce, Dedup-Map, Dedup-Groupby and Dedup-Reduce. Unfortunately, this requires to collect the output at each step, increasing the time complexity of the algorithm. That is why a simple option enables to disable this log generation.

5 Experimental Results

Experiments were conducted on Databricks using a Community Optimized cluster (15.3 GB Memory, 2 Cores, 1 DBU) running 7.5 (includes Apache Spark 3.0.1, Scala 2.12) and on our local machines Intel i7 2.9GHz, (2 cores) 16Go 1600Mhz DDR3 RAM. Attempts were made to launch AWS clusters on Rosetta Hub but this was not possible.

The data provided on the duration of the processes incorporates an error margin due to the memory and CPU allocation done by DataBricks and our operating system. Running the experiments several times and averaging the results will decrease the variance induced by the measurement errors.

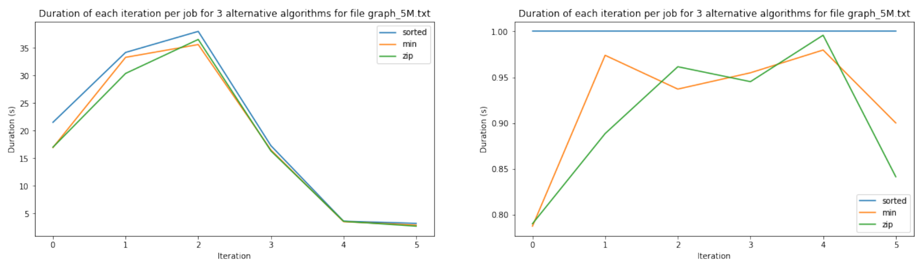
The readily available Webgraph from the Google programming contest (2002) contains 875,713 nodes and 5 105 039 edges. Below is the benchmark results obtained with the four implementations on this graph using the Databricks cluster:

- Python RDD: 452.357 seconds
- Python DataFrame: 489.215 seconds
- Scala RDD: 1164.194 seconds
- Scala DataFrame: 1245.516 seconds

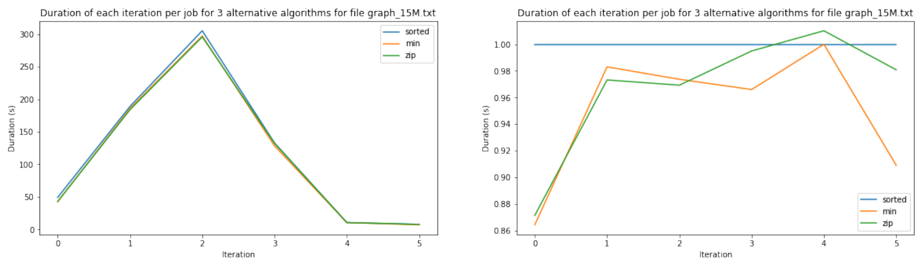
5.1 Different implementations from Section 3.3

We developed three concurrent versions of the algorithm on PySpark using RDDs. As developed in section 3.3, one version implemented secondary sorting before entering the it-reduce function, another used $numpy.argmin$ to isolate the minimum in the it-reduce function and the last implemented zip in order to avoid the for-loop of the it-reduce.

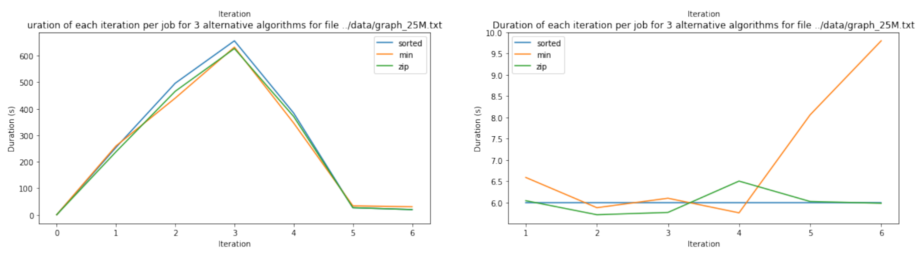
Figure 4 shows the differences in performance of these three algorithms on graphs of different sizes: 5 Megabytes, 15 Megabytes and 25 Megabytes.



(a) Duration of each execution in absolute and relative values using three different algorithms on a graph of size 5M.



(b) Duration of each execution in absolute and relative values using three different algorithms on a graph of size 15M.



(c) Duration of each execution in absolute and relative values using three different algorithms on a graph of size 25M.

Figure 4: Performance of three different algorithms on graphs of various sizes

For each sub-figure the graph on the left shows the average duration of each iteration in absolute value. The graph on the right shows the relative duration of each iteration with the algorithm using secondary sorting as baseline (value of 1).

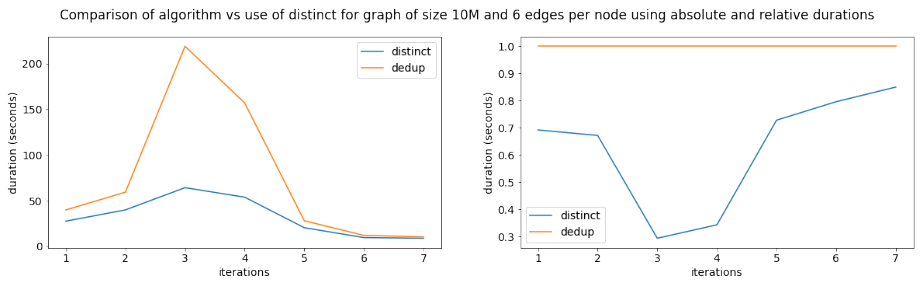
Below are the total average duration of the three algorithms on these three graphs:

- 5M graph:
  - secondary sorting: 117.78 seconds
  - argmin: 108.77 seconds
  - zip: 106.54 seconds
- 15M graph:
  - secondary sorting: 694.98 seconds
  - argmin: 671.84 seconds
  - zip: 673.48 seconds
- 25M graph:
  - secondary sorting: 1836.31 seconds
  - argmin: 1741.78 seconds
  - zip: 1747.33 seconds

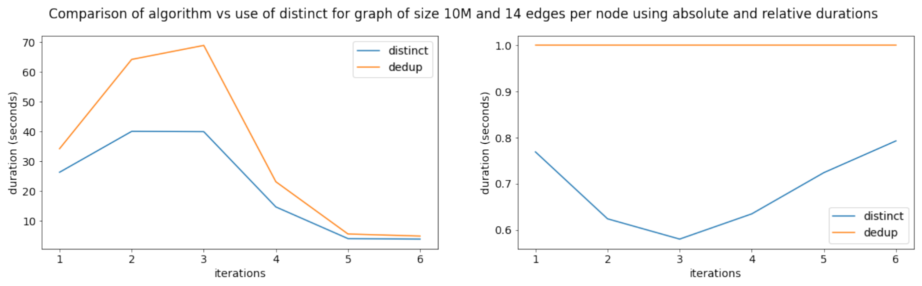
Both the total duration and each iteration duration show little evidence for a significant difference between the three algorithms. Nevertheless, the secondary sorting algorithm keeps performing a little worse than the two other algorithms, especially the one using $numpy.argmin$ to isolate the minimum. As expected from section 3.3 this highlights the slight difference between the complexity $\mathcal{O}(x * log(x))$ of a sorting algorithm compared to the $\mathcal{O}(x)$ complexity of $numpy.argmin$.

The following test shows the performance comparison between the original algorithm and an alternative using $distinct$. The original algorithm removes the duplicates in three steps. First, the edges are aggregated using $groupByKey$ and a single version of each edge is then unpacked to serve as input for the next iteration. This comparison has been realized using the Python RDD implementation on three graphs of similar sizes (10 Megabytes) but with different settings for the edges per node parameter, respectively 6, 14 and 22. Figure 5 shows the outcome of this comparison. For each sub-figure the graph on the left shows the average duration of each iteration in absolute value. The graph on the right shows the relative duration of each iteration with the algorithm using the original algorithm as baseline (value of 1).

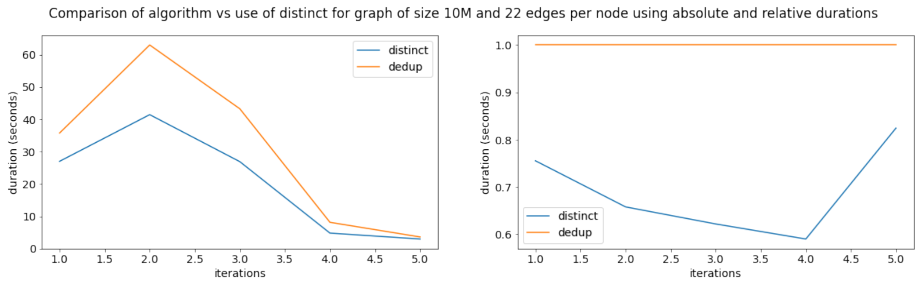
As expected in section 3.2, the implementation using $distinct$ is much more efficient than the one using the algorithm CCF-Dedup. This highlights the benefits of using Spark instead of the limited set of operations of Hadoop MapReduce. The use of $distinct$ enables to achieve a 20% to 70% performance improvement depending on the iteration and the edge per node setting.



(a) Duration of each execution in absolute and relative values using three different algorithms on a graph of size 5M.



(b) Duration of each execution in absolute and relative values using three different algorithms on a graph of size 15M.



(c) Duration of each execution in absolute and relative values using three different algorithms on a graph of size 25M.

Figure 5: Performance of three different algorithms on graphs of various sizes

5.2 Impact of the number of nodes and edges

Our graph generator requires the user to define the number of edges per node. Therefore, it is possible to set either the number of edges or the number of nodes and make the other parameter vary to study its influence on the performance of the algorithm. These tests have been realized with the version of the algorithm using $argmin(x)$ in IT-Reduce instead of the original version using secondary sorting. Moreover, the original version of the Dedup algorithm has been kept to stick to the original algorithm.

5.2.1 Setting the number of edges

We ran the algorithm on three graphs of 10Mb with different settings for the edges per node parameter. These three graphs have similar numbers of edges but very different numbers of nodes. Below are the exact features of the three files:

- graph_10M_3.txt: 266,282 nodes, 796,330 edges, 5.98 edges per node
- graph_10M_7.txt: 122,600 nodes, 860,444 edges, 14.04 edges per node
- graph_10M_11.txt: 81,283 nodes, 894,141 edges, 22.00 edges per node

Figure 6 shows the total duration of each run using the *dedup* algorithm and the *distinct* function. We notice that there is a negative relationship between the number of edges per node and the time complexity. As the number of edges remains constant, this is equivalent to a positive linear relationship between the number of nodes and the time complexity. The more nodes the more time it takes to execute even if the number of edges remains constant. The gap is wider when the *dedup* algorithm is used which indicates that most of the difference comes from the *dedup* part of the algorithm. Figure 7 confirms that the *dedup* part and especially its *groupby* function is computationally expensive.

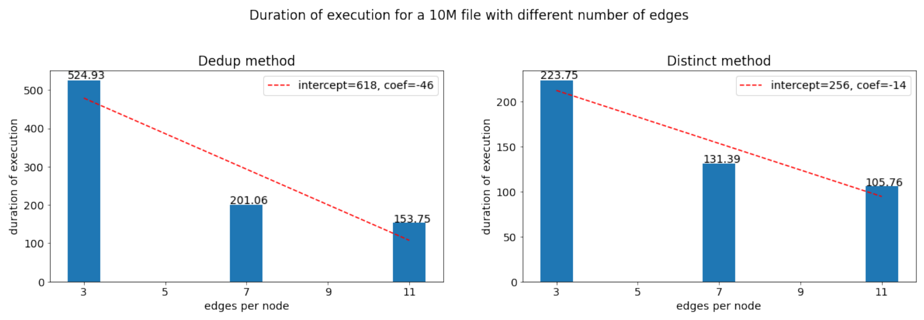


Figure 6: Duration of execution for a 10M file with different number of nodes.

Figure 8 shows the duration of each iteration. It highlights that the smaller the number of edges per node, the more iterations are required. Indeed, in the case when the number of edges per node is small, each node has less probability to be directly connected to the minimum of the cluster. Therefore, the algorithm needs more iterations to redirect all the connections towards the minimum of each cluster. Moreover, we notice that iteration 3 is especially long for the run with 6 edges per node.

Finally, Figure 9 analyses the duration of each *groupby* of the *dedup* phase as a function of the sizes of its input and output. We can see that the duration of the groupby relies mostly on the size of the output and very little on the size of its input as highlighted by a comparison of iterations 3 and 4. Nevertheless, a traditional SQL groupby

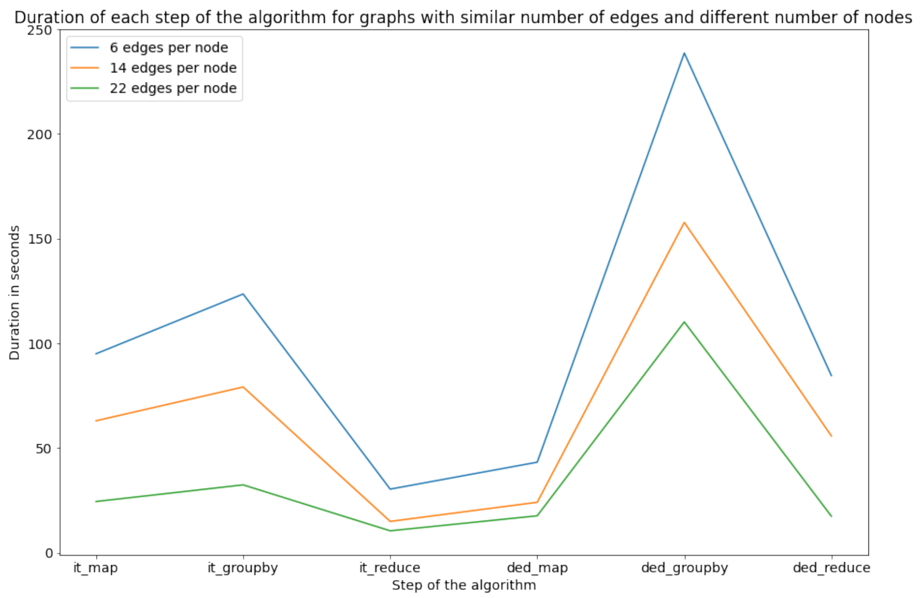


Figure 7: Duration of each step of the algorithm for graphs with similar number of edges and different number of nodes

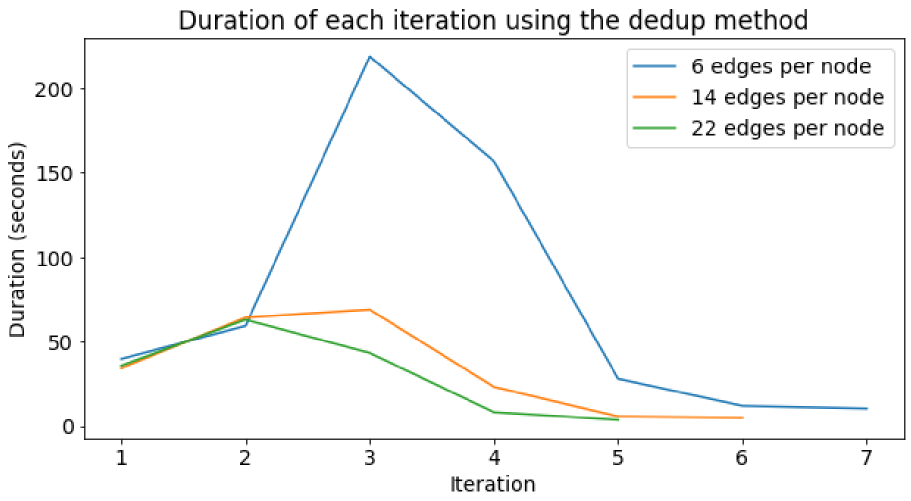


Figure 8: Duration of each step of the algorithm for graphs with similar number of edges and different number of nodes.

operation has a complexity $\mathcal{O}(sizeofinput)$. This difference comes from the parallelization of the operation. Spark parallelizes the *groupby* among the different cores but needs to join the output of each worker at the end to group the keys detected by several workers at the same time. On iteration 3, there is little difference between the input and the output. Therefore, the parallel operation reduces marginaly the size of the RDD and the final $groupby$ takes a large RDD as input. On iteration 4, the output is much smaller than the input. Therefore, the parallel operation is likely to aggregate many duplicates and the final $groupby$ takes as input a much smaller RDD. That is why the $groupby$ operation takes much more time on iteration 3 than on iteration 4.

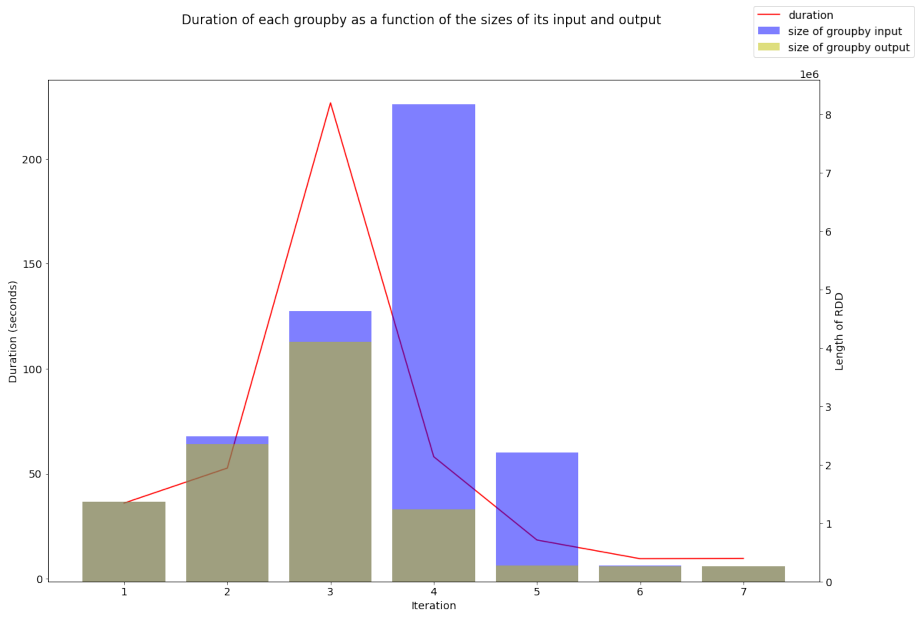


Figure 9: Duration of each groupby as a function of the sizes of its input and output.

To conclude, the differences in performances between graphs with different densities of edges per node come from the fact that the algorithm converges more easily to connections towards the minimum of a cluster when nodes have many connections. Indeed, it will need fewer intermediary steps to reach the minimum of a cluster from any node. On the opposite, a less dense graph results in longer paths to connect a node to the minimum of its cluster. Therefore, the graph has more "unique connections" at each step which increases a lot the duration of the $groupbys$.

5.2.2 Setting the number of nodes

We ran the algorithm on three graphs of different sizes, making sure that the number of nodes remained similar across the graphs. Below are the exact features of the three files:

- graph_4M_3.txt: 115,426 nodes, 347,334 edges, 6.02 edges per node

- graph_10M_7.txt: 122,600 nodes, 860,444 edges, 14.04 edges per node
- graph_15M_11.txt: 117,710 nodes, 1,298,544 edges, 22.06 edges per node

Figure 10 shows the total duration of each run and its step by step subdivision. When the number of nodes remains constant, the more edges the longer the execution time. It is interesting to notice that the steps it_reduce and ded_map take exactly the same exact amount of time in the three graphs as both operations have a time complexity $\mathcal{O}(number\_of\_nodes)$. The operation it_map takes more time with more edges as it has a complexity $\mathcal{O}(number\_of\_edges)$ (it only duplicates the edges by swapping the nodes).

Figure 11 explains the difference in the duration of the $it\_groupby$ step. The length of the output of $it\_groupby$ is exactly of the size number of nodes, which remains constant across iterations and among the three graphs tested here. Therefore, the difference in computation time comes from the difference in the size of the input, which depends on the number of edges (especially at the first iteration, it exactly equals $2 * numberofedges$).

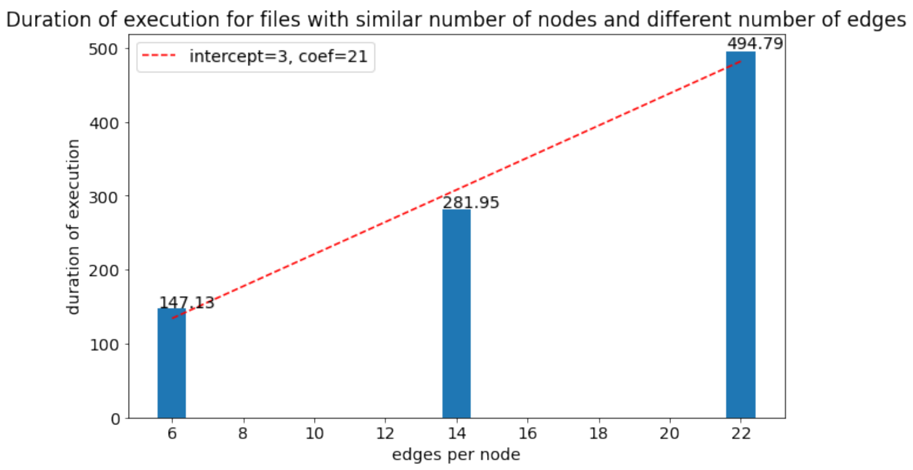
The differences in the performance of the $dedup$ algorithm have the same explanation as in Section 5.2.1

5.3 PySpark Implementations

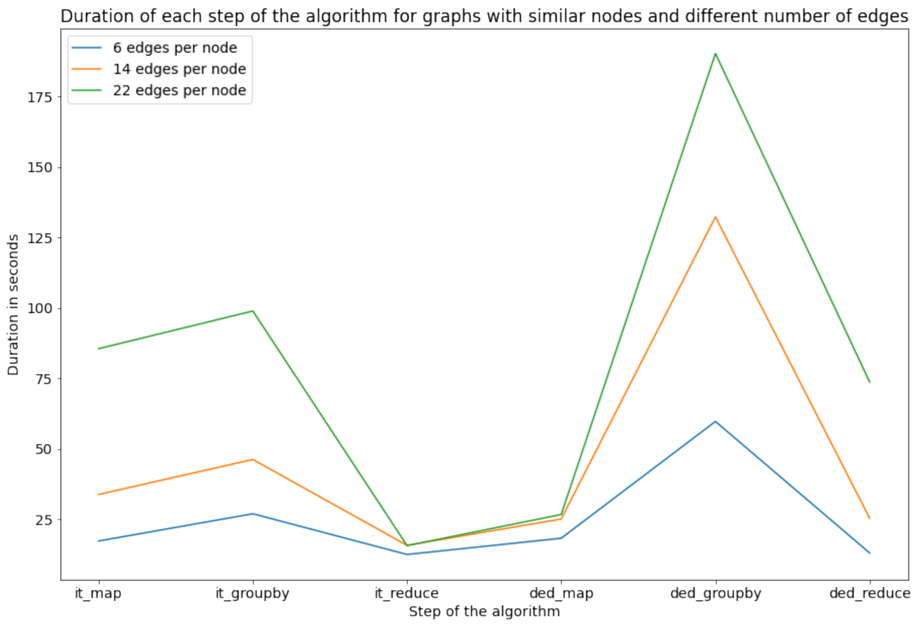
Both RDD and Dataframe implementations in Python have been tested on graphs of increasing sizes. Below is the composition of each of these graphs:

- graph_10M.txt: 166,950 nodes, 834,754 edges, 5 edges per node
- graph_20M.txt: 316,202 nodes, 1,581,012 edges, 5 edges per node
- graph_30M.txt: 466,559 nodes, 2,332,796 edges, 5 edges per node
- graph_40M.txt: 616,617 nodes, 3,083,086 edges, 5 edges per node
- graph_50M.txt: 764,818 nodes, 3,824,094 edges, 5 edges per node
- graph_60M.txt: 914,619 nodes, 4,573,098 edges, 5 edges per node
- web-google.txt: 875,713 nodes, 5,105,039 edges

Figure 12 shows the comparison of the duration of each iteration for the graph Web-Google. For this specific file, the RDD implementation is slightly faster in spite of a slow initialization. The size of the graph indeed does not justify the use of dataframes, which are more powerful on bigger files.



(a) Overall execution time for graphs with similar number of nodes and different number of edges.



(b) Step by step comparison of execution times for graphs with similar number of nodes and different number of edges.

Figure 10: Graphs with similar number of nodes and different number of edges

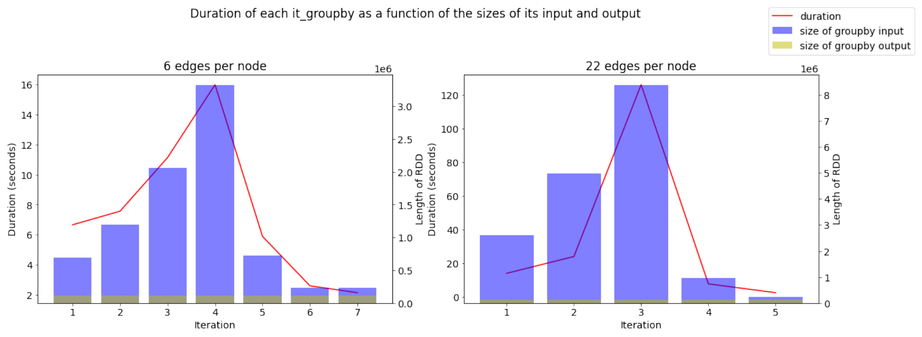


Figure 11: Duration of each it_groupby as a function of the sizes of its input and output.

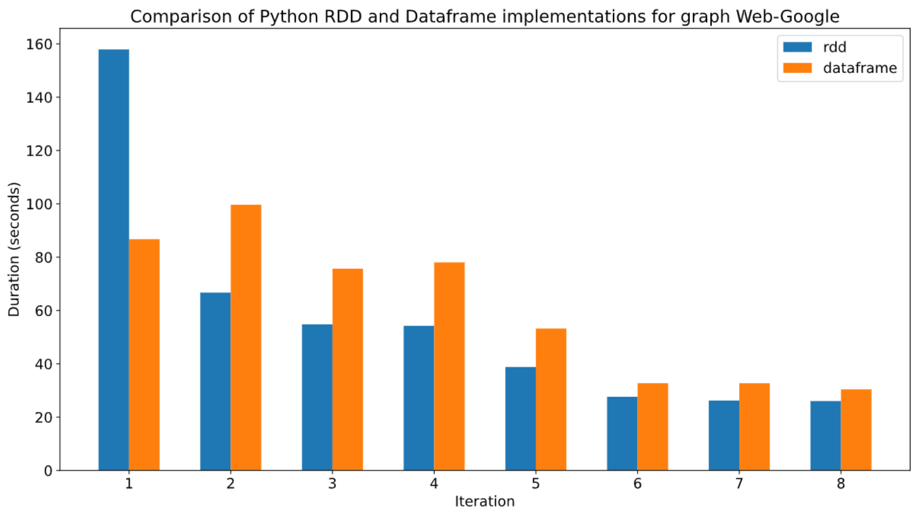


Figure 12: Comparison of Python RDD and Dataframe implementations for graph Web-Google.

5.4 Scala Implementations

Both RDD and Dataframe implementations in Scala have been tested on graphs of increasing sizes. Below is the composition of each of these graphs:

- graph_10M.txt: 166,950 nodes, 827,663 edges, 10 edges per node
- graph_20M.txt: 316,202 nodes, 1,577,341 edges, 10 edges per node
- graph_40M.txt: 616,617 nodes, 3,075,032 edges, 10 edges per node
- graph_50M.txt: 764,818 nodes, 3,824,241 edges, 10 edges per node
- graph_60M.txt: 914,619 nodes, 4,573,098 edges, 10 edges per node
- web-google.txt: 875,713 nodes, 5,105,039 edges

Figure 13 shows the comparison of the duration of each iteration for the graph Web-Google. It highlights that both implementations have very similar complexities and achieve similar performances. Comparing figure 13 with figure 12, its equivalent in Scala, we notice that the distribution of the duration among the iterations is very different between the two implementations. In Scala, iteration 3 and 4 take much more time than the others, while the duration of each iteration decreases in PySpark.

Figure 14 compares the performance of both algorithms on graphs of increasing number of edges and nodes. It shows a linear complexity when the number of nodes and edges increase in parallel. Thanks to the replacement of the $dedup$ algorithm by the $.distinct()$ function, the RDD version performs better and scales better than the Dataframe version.

6 Setup

We executed our algorithm in a notebook that runs on a cluster hosted on DataBricks. This notebook is provided in the Github repository supporting this project, in the subsection ”notebook”. Below are described the few steps of the execution:

- **Downloading resources**: As the iPython notebook supports UNIX shell commands and as a UNIX shell is installed by default on the Databricks’ clusters, we are able to download directly our environment using the *git clone* command.
- **Compiling the graphs generator**: As Databricks’ clusters supports Makefile, we are able to compile our generator of graphs directly in the cluster.

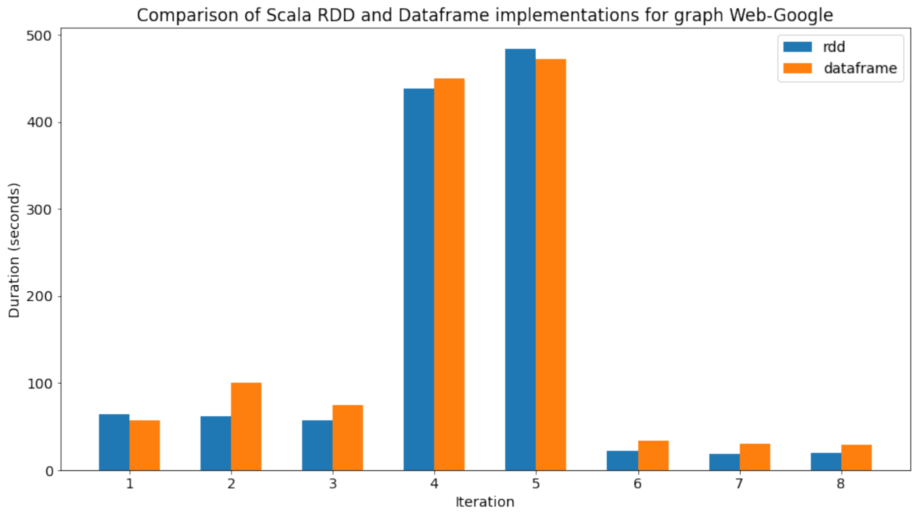


Figure 13: Comparison of Scala RDD and Dataframe implementations for graph Web-Google.

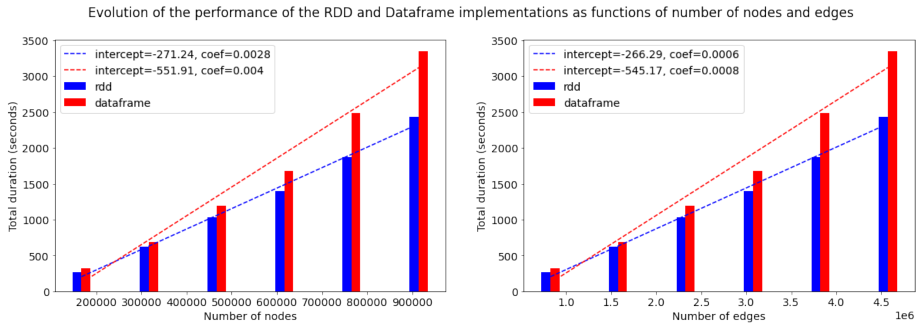


Figure 14: Evolution of the performance of the RDD and Dataframe implementations as functions of number of nodes and edges.

- **Defining a list of sizes for graphs to be tested**: The user inputs a dictionary in which keys are the sizes of graphs to be generated and values are the expected number of edges per node.
- For each entry of user input:
  - The graph is generated and stored on the cluster.
  - The file is converted to the dbfs file system.
  - The graph is processed as described in Section 3.
- **Parsing logs**: *SparkLogger* class parses the logs of the cluster. The pre-processed logs are then pushed to the repository.
- **Analytics**: Outside of the Databricks cluster, the pre-processed logs are analyzed and data visualisation is performed.

7 Conclusion

This work presents the implementation of an algorithm finding connected components in graphs using different APIs on Spark. The performance analysis shows that the PySpark implementations perform way better than their equivalent in Scala on graphs up to 1 million nodes and 5 million edges. The Scala implementations seem to have a linear scalability on this range of graphs, nevertheless, it is insufficient to conclude on the scalability of these algorithms on graphs wit different orders of magnitude. The performance of the PySpark implementations shows an inflexion point after 1 million nodes, as the RDD implementation starts performing better than the DataFrame version. Even if the scalability seemed to be linear up to this point, we also lack evidence to conclude on graphs of a bigger scale. More powerful clusters than the ones we had at our disposal would be necessary to conduct such an experiment.

References

[1] Hakan Kardes et al. “CCF: Fast and scalable connected component computation in MapReduce”. In: Feb. 2014, pp. 994–998. ISBN: 978-1-4799-2358-8. DOI: 10.1109/ICCNC.2014.6785473.

[2] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. “Pegasus: mining peta-scale graphs”. In: *Knowledge and information systems* 27.2 (2011), pp. 303–325.

[3] Thomas Seidl, Brigitte Boden, and Sergej Fries. “Cc-mr–finding connected components in huge graphs with mapreduce”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2012, pp. 458–473.

[4] *Apache Spark Jobs hang due to non-deterministic custom UDF*. `community.cloud.databricks.com/login.html#notebook/480503634711428/command/480503634711439`. Accessed: 2021-03-28.

[5] Antti Haapala. *Python's len implementation in C*. `https://stackoverflow.com/questions/53988132/explanation-of-c-implementation-pythons-len-function/53988387#53988387`. Accessed: 2021-03-08.

A Python Implementations

A.1 RDD Implementation

```python
def processGraphSorted(filepath):

  def reduce_ccf_sorted(x):
    key = x[0]
    values = x[1]
    min_value = values.pop(0)
    ret = []
    if min_value < key:
      ret.append((key, min_value))
      for value in values:
        accum.add(1)
        ret.append((value, min_value))
    return (ret)


  text_file = sc.textFile(filepath)
  text_file_split = text_file.map(lambda x: x.split())
  input = text_file_split.map(lambda x: (int(x[0]), int(x[1])))

  accum = sc.accumulator(1)
  while accum.value > 0:

    accum.value = 0
    print("Start loop ", accum.value)

    # CCF-Iterate
    it_map = input.flatMap(lambda x: ((x[0], x[1]), (x[1], x[0])))
    it_groupby = it_map.groupByKey().mapValues(lambda x: sorted(x))
    it_reduce = it_groupby.flatMap(lambda x: reduce_ccf_sorted(x))

    # CCF-Dedup
    ded_map = it_reduce.map(lambda x: ((x[0], x[1]), None))
    ded_groupby = ded_map.groupByKey().mapValues(list)
    input = ded_groupby.map(lambda x: (x[0][0], x[0][1]))

    collected = input.collect()
    print("End loop ", accum.value)

  print("Processed file")
```

Source Code 2: Python RDD Implementation.

A.2 DataFrame Implementation

```python
def processGraphDF(filepath):

  def reduce_ccf(key, values):
    min_value = values.pop(values.index(min(values)))
    ret = {}
    ret[key] = min_value
    if min_value < key:
      for value in values:
        acc.add(1)
        ret[value] = min_value
    else:
      ret = None
    return ret

  reducer = F.udf(lambda x, y: reduce_ccf(x, y), MapType(IntegerType(), IntegerType()))
  .asNondeterministic()

  schema = StructType([
      StructField("key", IntegerType(), True),
      StructField("value", IntegerType(), True)])

  df = spark.read.format('csv').load(filepath, headers=False, delimiter='\t', schema=schema)

  df = df.na.drop()
  acc = sc.accumulator(1)
  loop_counter = 1

  while acc.value != 0:
    acc.value = 0
    print(f"----------\nStart loop at {datetime.now()}, accumulator value is {acc.value}")

    # CCF-Iterate
    df_inverter = df.select(F.col('value').alias('key'), F.col('key').alias('value'))
    df = df.union(df_inverter)
    df = df.groupBy('key').agg(F.collect_list('value').alias('value'))
    df = df.withColumn('reducer', reducer('key', 'value')).select('reducer')
    df = df.select(F.explode('reducer'))
    df = df.na.drop()

    # CCF - Dedup
    df = df.distinct()

    collected = df.collect()
    df = spark.createDataFrame(sc.parallelize(collected), schema)
    print(f"End loop at {datetime.now()}, final value is {acc.value}")
    loop_counter += 1
  print(f"----------\nProcessed file at {datetime.now()}\n----------")
  return df
}
```

Source Code 3: Python DataFrame Implementation.

B Scala Implementations

B.1 RDD Implementation

```scala
def processGraphSorted(filepath: String): Unit = {

  def reduce_ccf_sorted(key: Int,
                        values: ListBuffer[Int],
                        accum: LongAccumulator):
                        ListBuffer[(Int, Int)] = {
    val min_value = values.remove(0)
    var ret = new ListBuffer[(Int, Int)]()
    if (min_value < key) {
      ret += ((key, min_value))
      accum.add(values.length)
      for (i <- 0 until values.length) {
        ret += ((values(i), min_value))
      }
    }
    return ret
  }

  val text_file = sc.textFile(filepath).filter(x => !(x contains "#"))
  val text_file_split = text_file.map(_.split("\t"))
  var input = text_file_split.map(s => (s(0).toInt, s(1).toInt))

  var accum = sc.longAccumulator("Accumulator")
  accum.add(1)
  while (accum.value > 0) {
    accum.reset()
    // CCF-Iterate
    val it_map = input.flatMap(x => List(List(x._1, x._2), List(x._2, x._1)) )
    val it_map_tuple = it_map.map(x => (x(0), x(1)))
    val it_groupby = it_map_tuple.groupByKey()
                                .mapValues(x => x.to[ListBuffer].sorted)
    val it_reduce = it_groupby.flatMap(x => reduce_ccf_sorted(x._1, x._2, accum))
    // CCF-Dedup
    val ded_map = it_reduce.map(x => ((x._1, x._2), None))
    val ded_groupby = ded_map.groupByKey()
    input = ded_groupby.map(x => (x._1._1, x._1._2))
    val ret = input.collect()
  }
}
```

Source Code 4: Scala RDD Implementation.

B.2 DataFrame Implementation

```scala
def graphsDF(filestore_name: String, df_logs: DataFrame, logs:Boolean): DataFrame = {

  def reduce_ccf(key: Int, values: Seq[Int]): ListBuffer[(Int, Int)] = {
    val list_values = values.to[ListBuffer]
    val min_value = list_values.remove(list_values.indexOf(list_values.min))
    var ret = ListBuffer[(Int, Int)]()
    if (min_value < key) {
      ret += ((key, min_value))
      acc.add(list_values.length)
      for (i <- 0 until list_values.length) {
        ret += ((list_values(i), min_value))
      }
    }
    return ret
  }

  var acc = sc.longAccumulator("accumulator")
  val schema = StructType(List(StructField("key", IntegerType, true),
                                StructField("value", IntegerType, true)))
  var df_logs_ret = df_logss
  var df = spark.read.format("csv").option("header", "false").option("delimiter", "\t")
      .schema(schema)
      .load(filestore_name)
  df = df.na.drop()

  val reducer = udf {(x: Int, y: Seq[Int]) => reduce_ccf(x, y)}.asNondeterministic()
  var extracted_val = 1L

  while (extracted_val != 0) {
    acc.reset()
    /* CCF-Iterate */
    val df_inverter = df.select(col("value").alias("key"), col("key").alias("value"))
    val df_it_map = df.union(df_inverter)
    val df_it_groupby = df_it_map.groupBy("key").agg(collect_list("value").alias("value"))
    var df_it_reduce = df_it_groupby.withColumn("reducer", reducer(col("key"), col("value")))
    df_it_reduce = df_it_reduce.select(explode(col("reducer")))
    df_it_reduce =df_it_reduce.na.drop()
    df_it_reduce = df_it_reduce.select(col("col._1").alias("key"), col("col._2").alias("value"))
    /* CCF - Dedup */
    var df_dedup = df_it_reduce.dropDuplicates()
    df = df_dedup.select(col("key"), col("value"))s
    df = spark.createDataFrame(sc.parallelize(df.collect()), schema)
    extracted_val = acc.value
  }
  return df_logs_ret
}
```

Source Code 5: Scala DataFrame Implementation.