Finding Connected Components in a Graph

Skander Adam Afi & Mamadi Keita

February 2026

Contents

List of Figures

Listings

1 Introduction

A graph is characterized by its nodes and edges, but more specifically, multiple methods are implemented to help identify all the components connected inside. One of those methods is Connected Component Computation (CCF), which operates through four steps; beginning with edge pairs of a defined source and distance, the algorithm iteratively propagates minimum node IDs to neighbors, only stopping where there are no changes to be done until all nodes map to the smallest ID in each component [1].

It is possible to translate this language into Spark code to understand the workings behind the CCF and visualize some important statistics in terms of searching for components. In this report, we will describe our adopted solution, how it was implemented, analyze it in terms of visualizations, and outline all strengths and weaknesses from what we can see.

2 Implementing the CCF with Spark

2.1 Description

To help understand how the CCF operates, we will implement it in Spark and evaluate its performance through:

- **PySpark:** It is the Python API for Apache Spark [4] and allows us to use Spark with Python for implementations such as CCF; it helps that it is easier to use, whereas Apache runs in clusters.
- **Scala:** Scala is Apache Spark's own programming language in which it runs. It is often used in data engineering from small scripts to large multi-platform applications [3].

To help understand how CCF operates in both Python and Scala languages, the following section will offer descriptions on how it is implemented with RDD [5] (a fault-tolerant collection of elements that can be operated in parallel) and DataFrame (stores data similar to a structured database) [2]

2.2 PySpark Implementation

After installing PySpark and the related packages required to code and visualize, we initialize the Spark session and generate a synthetic graph that we then convert to undirected, then transform into RDD format with the "map" function.

2.2.1 RDD

We start by initializing the iterations and number of components needed, then setting the algorithm to run while the changes are happening. Each grouped node and its list of neighbors are grouped and extracted before the minimum number of IDs is detected.

Listing 1: Propagation of Minimum Node Identifiers Across Neighboring Nodes

```
def propagate(node_neighbors):
    node, neighbors = node_neighbors
    neighbors = list(neighbors)

    if not neighbors:
        return []

    min_id = min([node] + neighbors)

    results = []

    if min_id < node:
        results.append((node, min_id))
        for n in neighbors:
            if n != min_id:
                results.append((n, min_id))

    return results
```

Every ID that is smaller than the node is propagated to its neighbors as the current node is updated, and all changes within the graph are detected as it is updated. The total number of components, when counted, is 22 in 2 iterations.

2.2.2 DataFrame

The graph is first initialized as a DataFrame and converted accordingly, as the minimum is computed and propagated across the graph as "new-df", which replaces the old version across a number of iterations.

Listing 2: Iterative Graph Update and Convergence Check Using DataFrame Transformations

```
    min_df = graph.groupBy("src") \
             .agg(spark_min("dst").alias("min_id"))
    new_df = graph.join(min_df, "src") \
             .filter(col("min_id") < col("src")) \
             .select(col("src"),
                     col("min_id").alias("dst")) \
             .distinct() \
             .cache()
diff = new_df.subtract(graph)
changed = diff.count() > 0

graph.unpersist()
graph = new_df
```

2.3 Scala Implementations

Scala is a slightly more complicated case in that it requires a Java application of 8/11/21, since it cannot run directly on Python notebooks. The project is compiled and built with "sbt", then run with Spark.

2.3.1 RDD

After all necessary imports, the graph is first initialized as each line is converted into a tuple. The "var" and "val" methods are used to define the methods and variables. Like the RDD within PySpark, the "updated" function is defined as the minimum node ID among both nodes and neighbors and updates them with the new ID, before removing all duplicate edges to easily cache the new RDD.

Listing 3: Minimum Node Identifier Propagation Step Using RDD Transformations

```
  // Propagate minimum IDs
    val updated = grouped.flatMap { case (node, neighbors) =>
      val neighborList = neighbors.toList

      if (neighborList.isEmpty) {
        Seq.empty[(Long, Long)]
      } else {
        val minId = (node :: neighborList).min

        if (minId < node) {
          val results = ArrayBuffer[(Long, Long)]()
          results += ((node, minId))

          for (n <- neighborList if n != minId) {
            results += ((n, minId))
          }
          results.toSeq
        } else {
          Seq.empty[(Long, Long)]
        }
      }
    }.distinct().cache()
```

The graph is then compared with the old versions, stopping only when there are no changes.

2.3.2 DataFrame

We proceed in the same way with the DataFrame implementation, in which the file is converted into a DataFrame and converting both src and dst into LongType as invalid entries are filtered. After initializing the undirected graph and removing all duplicates, we implement the CCF algorithm with the exact same functionalities, as all duplicates are removed within this stage as well.

Listing 4: DataFrame-Based Minimum Neighbor Computation and Graph Update (Scala)

```
  // Find minimum neighbor for each node
    val minDF = graph.groupBy("src")
      .agg(min("dst").alias("min_id"))
```

```
      // Create new edges where min_id < src
      val newDF = graph.join(minDF, "src")
        .filter(col("min_id") < col("src"))
        .select(
          col("src"),
          col("min_id").alias("dst")
        )
        .distinct()
        .cache()

      // Check for changes
      val diff = newDF.except(graph)
      changed = diff.count() > 0

      // Replace old graph
      graph.unpersist()
      graph = newDF

      val elapsedTime = (System.currentTimeMillis() - startTime) / 1000.0
      println(f"Iteration time: $elapsedTime%.2f seconds")
    }

    (graph, iteration)
  }
```

Like in the RDD, all duplicates are removed.

3 Experimental Analysis: Strengths & Weaknesses

3.1 Experimental Analysis Implementation

To understand the performance of CCF across both Python and Scala, a series of experimental analysis implementations is set in motion to discover how well each code performs for a set of increasing graphs.

3.1.1 Python

For both RDD and DataFrame, we implement a synthetic graph generator; RDD requires a parallelize function across multiple database dimensions, while "DataFrame" is used to convert the graph accordingly. For each case, we run the algorithm and evaluate its metrics, notably the number of iterations to find said components, and the total time taken from the start.

Listing 5: DataFrame and RDD Comparaison across Python

```
if verbose:
    print(f"  RDD Iteration {iteration}...", end=" ")
    total_time = time.time() - total_start
```

Both versions are tested across graphs of increasing size that range from 1000 et 50000 with an average degree of ten nodes.

Listing 6: DataFrame and RDD Comparaison across Python

```
Generated RDD graph: 1,000 nodes, 9,948 edges
Generated DataFrame graph: 1,000 nodes, 9,948 edges

--- RDD Implementation ---
  RDD Iteration 1... 0.59s
  RDD Iteration 2... 0.56s
  RDD Iteration 3... 0.57s
  RDD Iteration 4... 0.62s
Total: 2.34s, Iterations: 4, Components: 999

--- DataFrame Implementation ---
  DataFrame Iteration 1... 0.15s
Total: 0.15s, Iterations: 1, Components: 261

--- Comparison ---
Speedup: 16.00x (DataFrame is faster)
Time difference: 2.19s

================================================================================
Testing graph with 5,000 nodes (avg degree: 10)
================================================================================

Generating graphs...
Generated RDD graph: 5,000 nodes, 49,918 edges
Generated DataFrame graph: 5,000 nodes, 49,918 edges

--- RDD Implementation ---
  RDD Iteration 1... 0.50s
```

```
  RDD Iteration 2... 0.49s
  RDD Iteration 3... 0.51s
  RDD Iteration 4... 0.55s
Total: 2.04s, Iterations: 4, Components: 4,999

--- DataFrame Implementation ---
  DataFrame Iteration 1... 0.18s
Total: 0.18s, Iterations: 1, Components: 1,306

--- Comparison ---
Speedup: 11.19x (DataFrame is faster)
Time difference: 1.86s
```

Across RDD, it takes four to five iterations and an estimated average of two to three seconds, but when using DataFrame, the speed is exponentially reduced from 1 second to 0.37 in total.

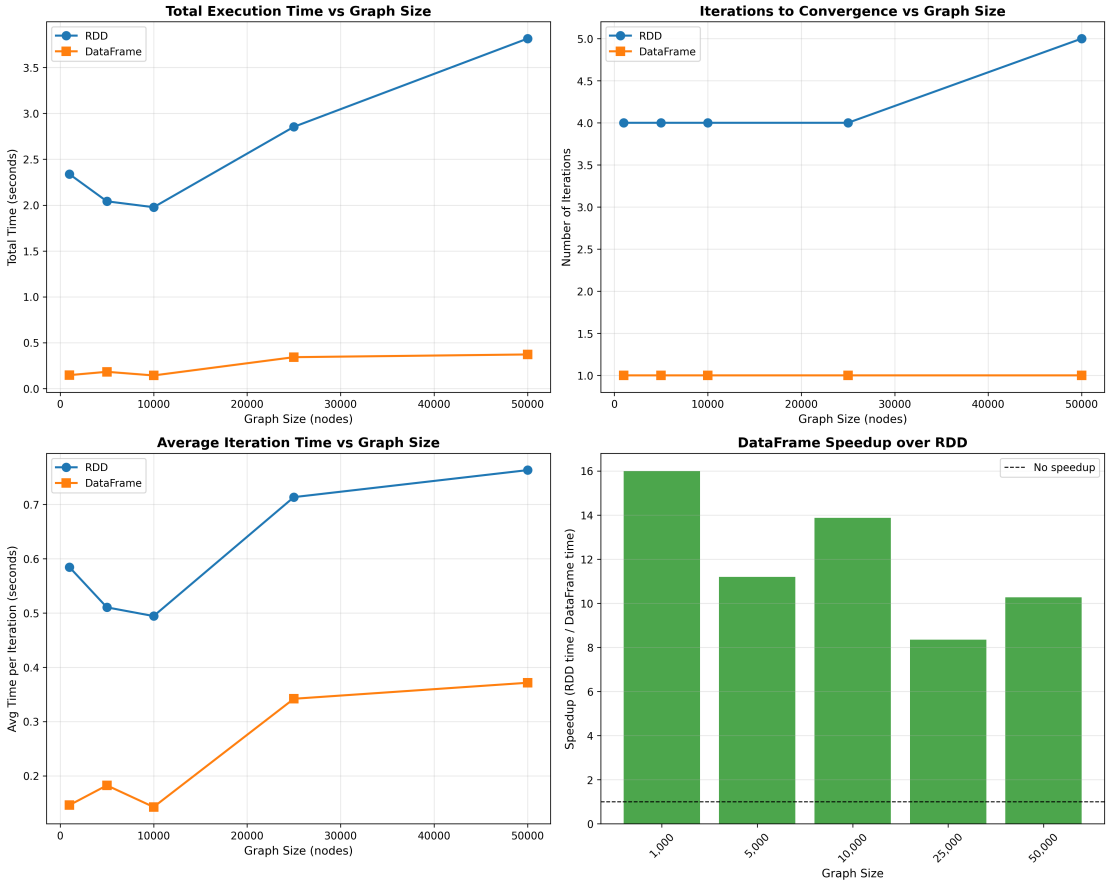

Figure 1: RDD and DataFrames: Comparisons across Iteration Time and Speedups

In turn, however, the number of components when using RDD is greater than that of DataFrame; RDD, despite having a slower execution time, offers a more correct way to find components, as seen from the higher iteration-to-convergence rate.

3.1.2 Scala

Across Scala, we implement those techniques as well; after defining the exact same graph sizes and synthetically generating each, we run the RDD and DataFrame experiments in Scala languages, but with the added average iteration time calculator by accumulating the number of iteration times and dividing them over the number of iterations.

Listing 7: Average Time Iteration method

```
    val totalTime = (System.currentTimeMillis() - totalStartTime) / 1000.0
    val avgIterTime = iterationTimes.sum / iterationTimes.length
```

The results show a slight modification.

Table 1: Connected Components Performance (Scala)

Size	Edges	Impl.	Iter	Total Time (s)	Avg Iter (s)	Comp. Count
1000	9962	RDD	4	1.032	0.258	999
1000	9962	DataFrame	1	1.605	1.605	257
5000	49986	RDD	4	0.823	0.206	4999
5000	49986	DataFrame	1	3.040	3.040	1261
10000	99964	RDD	4	0.865	0.216	9994

DataFrame across Scala operates on one iteration compared to the 4 found in RDD. However, it not only generates less components but takes longer than the RDD to generate the desired results; unlike the tradeoff within PySpark, RDD compensates for the number of iterations with a faster time and more results.

3.2 PySpark vs. Scala: Strengths and Weaknesses:

All statistics related PySpark and Scala are then loaded into CSV files to compare within both languages. Once this is done, both datasets are combined into a single one that displays all of them across a table.

Whereas both implementations show similar algorithmic behavior, they show a fair share of differences as well.

Table 2: Performance Comparison: PySpark vs Scala

Size	Edges	PySpark RDD	PySpark DF	Scala RDD	Scala DF	Scala/Py RDD	Scala/Py DF
1000	9948	2.337	0.146	–	–	–	–
1000	9962	–	–	1.032	1.605	–	–
5000	49918	2.042	0.182	–	–	–	–
5000	49986	–	–	0.823	3.040	–	–
10000	99944	1.977	0.143	–	–	–	–
10000	99964	–	–	0.865	3.841	–	–
25000	249946	2.854	0.342	–	–	–	–
50000	499946	3.816	0.371	–	–	–	–
50000	499970	–	–	1.680	4.445	–	–
100000	999966	–	–	2.400	15.996	–	–

Across the average execution time, PySpark is noticeably faster when using DataFrame, whereas using Scala for RDDs allows a more optimized time frame.

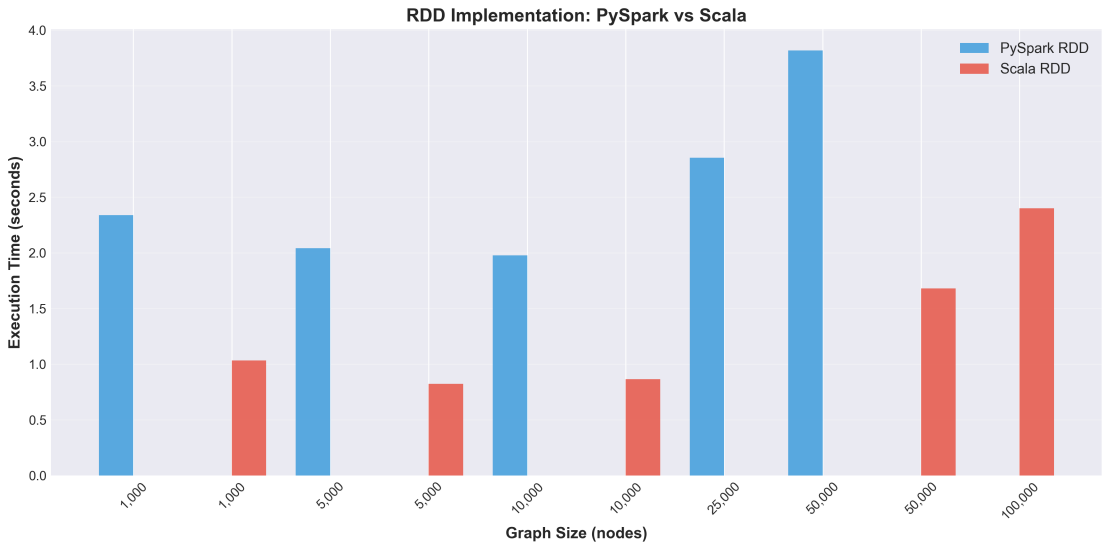

Figure 2: PySpark and Scala RDDs: a Comparison

The measures of this graph indicate that Scala is better equipped to handle graphs of increasing size; in fact, it runs natively on JVM, avoiding Python-JVM serialization overhead.

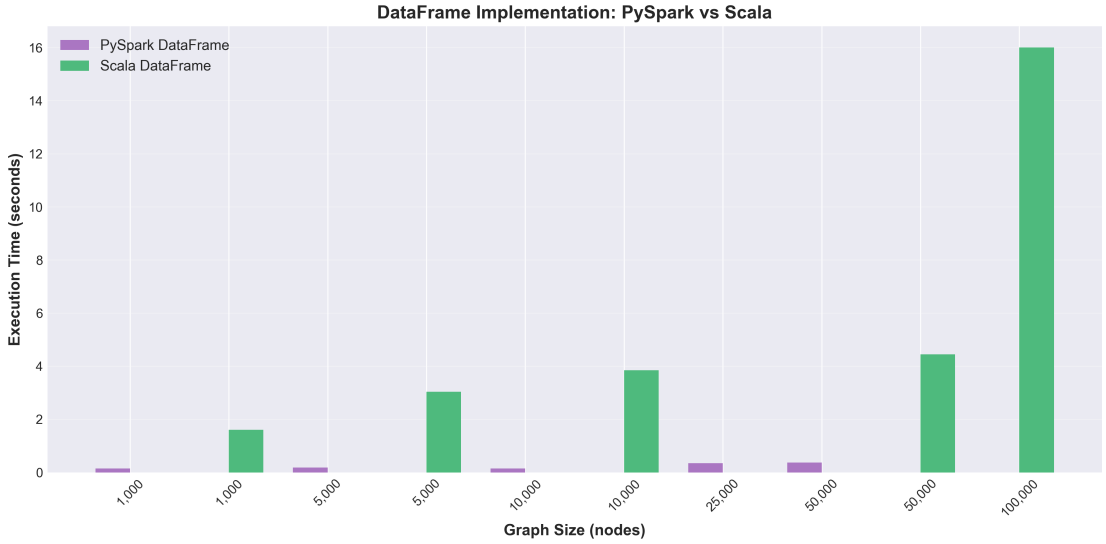

Figure 3: PySpark and Scala DataFrame Comparison

Conversely, PySpark is much slower than Scala when handling DataFrames;both DataFrame APIs benefit from Catalyst optimizer in both implementations, but in Python, it has additional overhead from Python interpreters and data conversions.

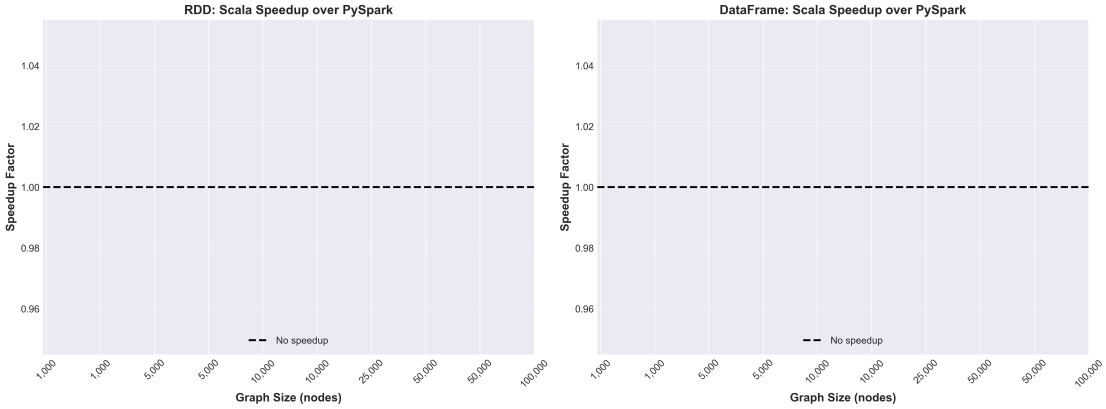

Figure 4: Speedup Comparison

The speedup comparison is similar for both; it illustrates their similar behavior and implementation.

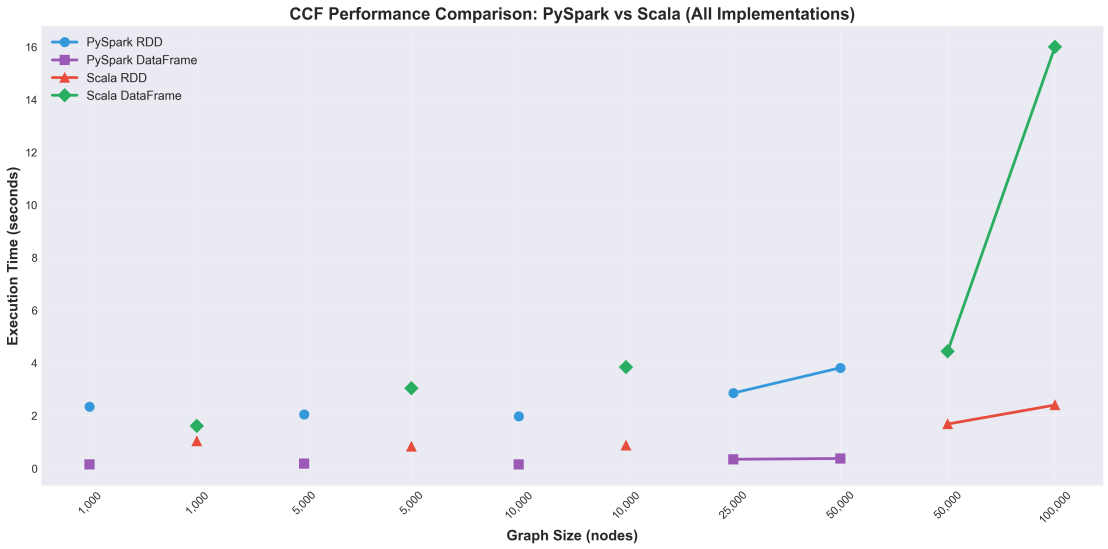

Figure 5: Global Comparison

Globally, we note that PySpark DataFrame offers a faster execution time and an easier development, allowing for a competitive performance when exploring new data. Scala provides maximum throughput in a shorter time frame when more performance and competitiveness are required.

4 Conclusion

In conclusion, PySpark and Scala offer varying implementations of the CCF algorithm, and each metric depends on the language used and the conditions required to achieve either maximum throughput or faster performance.

It is possible to extend the use of this algorithm within the real world; in vast commercial systems for example, large record linkages are useful to create records for a single person. It is also very useful for social network analysis on a global scale to determine all online users for distributed graph processing.

References

[1] Hakan Kardes, Siddharth Agrawal, Xin Wang, and Ang Sun. Ccf: Fast and scalable connected component computation in mapreduce. *Data Research, inome Inc.*, 2013.

[2] Pure Storage. RDD vs DataFrame: What's the difference? `https://blog.purestorage.com/purely-educational/rdd-vs-dataframe-whats-the-difference/`, 2024. Accessed: 2026-02-18.

[3] Scala Language. Scala programming language – official website. `https://www.scala-lang.org/`, 2026. Accessed: 2026-02-18.

[4] The Apache Software Foundation. Apache Spark python api documentation. `https://spark.apache.org/docs/latest/api/python/index.html`, 2026. Accessed: 2026-02-18.

[5] The Apache Software Foundation. Apache Spark — rdd programming guide. `https://spark.apache.org/docs/latest/rdd-programming-guide.html`, 2026. Accessed: 2026-02-18.